BDS Cの使い方

稲川幸則
渡辺　修　共訳

BDS

Compiler

工学図書株式会社

BDS　C の使い方

稲　川　幸　則
渡　辺　　　修　共訳

工学図書株式会社版

BDS C User's Manual

序　　文

BDS Cは，8080マイクロプロセッサ用のコンパクトで高性能なコンパイラであり，CP/Mシステムの下で動作する．また，Z-80や8085などのCPU上でも動作する．BDS Cの製作者であるLeor Zolman氏は，8080の限られた命令と，64Kバイトという今では少なくなってしまったメモリ・エリアに，コンパイラを移植してしまった！　しかもたった2つのコンパイラ・モジュールで，8080のコードを生成してしまうのである．それだけではない．このコンパイラは，コード・ジェネレーションのオプティマイザやその他の多くの機能を有しており，コンパイル速度も申し分ないものである．しかし，彼は現在に至るまでかなり苦労したようで，数十回のコンパイラのバージョン・アップがそれを物語っている．いずれにしても，8080系のCコンパイラで，これだけのパフォーマンスを持つものは他に類を見ないであろう．

コンパイラのパフォーマンスもさることながら，低価格ということも魅力の一つであり，かなりの数の人が，彼のコンパイラのユーザーである．そして彼らは，ユーザーズ・グループを結成し活発な動きを見せており，多くのアプリケーション・プログラムやユーティリティを発刊している．その数は，フロッピーディスケット数十枚に及ぶものである．

本書は，バージョン1.50と1.50aの追加インフォメーションを含めた，最終版のマニュアルの和訳である．原版の第1章で述べられている価格や新しい機能，そして古いコンパイラとの差異の節は省略し，実際の操作や関数の使用方法に重点を置いた．始めてコンパイラを使うときは，まず第4章に目を通していただきたい．この章は，カーニハン&リッチーの本の付録Aとの差異について述べてあるので，BDS Cの能力の概要を知ることができる．

また，BDS Cの能力増強を計るためのアセンブリ言語とのリンク，倍精度整数，浮動小数点パッケージ，そしてユーティリティとして通信プログラム，シンボリック・デバッガが付録に収録されている．

BDS Cの世界を楽しんで下さい！

昭和 59 年 6 月

著　者

目　　次

第 1 章　序　章

第 2 章　CRL 関数フォーマットと低レベル関数

第 4 章 The C programing Language の付録Aに関する註釈

付　録

第 1 章

序 章

Leor Zolman
BD Software
P.O Box 9
Brighton Massachusettes 02135
(617)782－0836

1.1 はじめに

本書は，ミニコンピュータ用高水準言語であるCを，プログラマーが個人的に利用するためのものであり，CP/M.OSを使用しているマイクロコンピュータのために特別に適合させてある．BDS Cの当初の設計目標はCのプログラマーが，構造的プログラムの開発を安定かつ**能率的**なペースで行なえるようにすることであった．最終的には，プログラマーがじっとがまんしなくても，コンパイラとリンカーは十分な速度で繰り返し，プログラムをコンパイルし，リンクし，実行するようになった．

作業を進めて行く上で，全プロセスを繰り返さなければならないようなつまらないエラーが，何分も後にあげられるといったようなことは全くない．
BDS Cを用いると，こういったエラーはたいていコンパイルを始めて数秒の

間に現われるし，プログラマーは，スローなコンピュータに向かってブツブツ文句を言うかわりに，**プログラミング**して時をすごすことができる．

1.2 目的と制限

BDS Cのコンパイラは，UNIX[1]のOSと関連してベル・ラボラトリーで独創的に開発されたCプログラミング言語[2]のサブ・セットの実現である．コンパイラそのものはCP/M[3]のOSが走る8080／Z80のマイクロコンピュータ・システムで実行され，CP/M上でか，ROMかRAMの任意のロケーションで実行可能なコードを生成する．(ターゲットマシーンのどこかに，ランタイムで必要なRAMがなければならないが.)

この企画の主な目的は，ミニコンピュータからマイクロコンピュータ環境にまでわたって基づいているUNIX OSのパワフルな構造的プログラミング原理を少し翻訳することであった．BDS Cは，コンパイラと最終応用との両方の利用のために，エレガントで能率的なヒューマン・インタフェイスに重点をおいて，CP/Mユーティリティやアプリケーションを開発する親しみやすい環境を供給する．

残念ながら，C言語の構造は，PDP-11[4]のハードウェアの特質に適応するようになっており,8080のハードウェアの特質に対して適応しない.PDP-11にとって自然な操作（自動記憶割付けを処理するとき，大変重要なインデックス付きや，非間接的アドレス指定のような）は，8080においてはむしろ非能率的な命令コードへと展開されることになる．このためBDS Cは，8080システムプログラミング言語としてPDP-11のUNIX Cのように急速に一般化することは

1）UNIXはベル研究所の登録商標である．

2）完全に言語を知るためにBrian W. KerninghanとDennis M. Ritchie (Prentice Hall.1978)によるCプログラミング言語を参照すること．本書はBDS Cの機能についての説明書であり，C言語の解説書ではない．(**注**：この本の和訳版は，共立出版株式会社から"プログラミング言語C-UNIX流プログラム書法と作法"という題名で発刊されている.)

3）CP/Mはデジタル・リサーチ社の登録商標である．

4）PDPはデジタル・イクイップメント社の登録商標である．

ないだろうと思われるが，有能なマイクロプロセッサが，現存する8ビットマシンと置き換わり，アプリケーション志向のアセンブリー言語プログラミングやヒストリ・ブックを放棄するに値する有能なCコンパイラが出現することは確実である．本書は，そのときのためのウォーミングアップと考えてもらいたい．

かいつまんで言えば，アセンブリー言語のプログラミングと比較したときにBDS Cの大きな矛盾は，開発環境の機構と理解を重視したため，空間的かつ時間的な実行時のオブジェクトコードの能率の低下を招いてしまったことである．しかし，ほとんどすべての教育上の，また，ほとんどのシステムのプログラミング・アプリケーションにとって，犠牲は益よりも少ないと思う．

1.3 システムの構成

BDS Cの必要とする実践的最小ハードウェア構成は，40K CP/M2.x のシステムである．パッケージに含まれるほとんどのサンプルプログラムは，（分割しないで）コンパイルし，48Kシステムで実行できる．

BDS Cは，ソーステキストを少しずつ読み込んでコンパイルするのではなく，**全ソース・グループを**一度にメモリーへロードし，メモリ内でコンパイルを行なう．これによって，前者のアルゴリズムに比べて極めて迅速にコンパイルができるのである．適当な大きさのソーステキストをコンパイルする上で，一番障害となるものは，現在のところソース・テキストの読み取りと，CRLファイルを書き出す時に含まれるディスクI/Oであるが，これらの操作は，標準CP/Mシステムが操作できるのと同じ位の速さで行なわれる．この企画での制限は，ソースファイルはコンパイルのためにメモリーへ入りきる大きさでなければならないということである．これは最初，あなたに悪い印象を与えるかもしれないが，実際には，Cのプログラムはトップレベルの一つの main 関数から生じている小さめの関数が多く集まったものである．それぞれの関数は，別々にコンパイルされ，別のコンパイルされた関数とともに，1つの完全なプログラムを形成する．

このように単一のプログラムは，多くのソースファイルに分けることができ，それぞれいくつかの関数を含んでいる．プログラムをいくつかのファイルへ区分化することによって，個々のソース・ファイルが有効なメモリーをオーバーフローすることを防ぐだけでなく，小さな変化にともなう再コンパイルの時間を最小限にとどめる．

1.4 コンパイラの使用法

1.4.1 コマンドとデータ・ファイル

主なBDS Cパッケージは次の4つの実行可能なコマンドで成り立っている．

CC.COM Cコンパイラ第1パス
CC2.COM Cコンパイラ第2パス
CLINK.COM Cリンカー
CLIB.COM Cライブラリ管理プログラム

それから，リンカーが通常必要とする3つのデータ・ファイル

C.CCC ランタイム初期化ルーチンとサブルーチン・モジュール
DEFF.CRL 標準（“デフォルト”）関数・ライブラリ
DEFF2.CRL もう1つのライブラリ関数

CC.COMとCC2.COMは，いっしょになって実際のコンパイラを形づくる．CCはディスクから与えられたソースファイルを読み取り，その上をかみ砕いて進み，メモリーの中に中間ファイルを残し，コンパイルを終えた後，自動的にCC2をロードし，出力[5]としてCRLファイルをつくる．CRL(C Relocatable

5）必要な場合，CCがつくり出した，中間ファイルは，ディスクへ書き込まれ，CC2によって別々に処理される．その場合，中間ファイルの拡張ファイル名として，CCIが用いられる．

の意味）ファイルは，特別な再配置可能フォーマットで8080マシーン・コードを生成する．

リンカー，CLINKは，“main”関数を含んだCRLファイルと付随して必要となる関数を指定された全てのCRLファイルから検索する（このとき，DEFF.CRL，DEFF2.CRLとDEFF3.CRLは自動的に検索される）．全ての関数がリンクされると，COMファイルを生成する．

CLIBプログラムは，CRLファイルの内容の処理を行なうものである．

1.4.2 コンフィグレーション

BDS Cのコマンドは，特別な設定の手順なしで，CP/Mシステムのもとで実行できるが，パッケージの融通性を増すためにユーザーが任意に設定できるようなコンパイラとリンカーのオプションがいくつかある．このサブセクションは，これらのオプションの各々とそれらの選び方を説明する．

1.4.2.1 CCとCLINKのコンフィグレーション

これらの修正を試みる前に，マスターディスクを安全に保管してあることを確認すること！（バックアップコピーを作成しておく）

CC.COMとCLINK.COMの各コマンド・ファイルの最初の部分に位置する特別なメモリーエリアによってコントロールされるいくつかの機能をユーザー設定できる．これらの設定を変えるためには，DDTかSIDを使用し，メモリーの中へCC.COMかCLINK.COMを読み込み，Sコマンドを使って，その変更を行ない，コントロールCを入力し，修正されたコマンドをCP/MのSAVEコマンドを使用して再びディスクへ書き込む．

修正されたバージョンをディスクにセーブするために，DDTかSIDによってプリントされた16進の“NEXT”アドレスを10進にして，**SAVE**コマンドへ与えなければならない．10進の数字へ変換するために，次のアルゴリズムを使用する．まず一番左の2つの16進数字を取って，等価の10進数を計算する（たとえば，3C80は3Cをもたらし，これは10進数で60である）．それから，一番右の2つの数字が00である場合のみ，1を引くと（たとえば，上の60は60のまま

である．というのは，3C80の一番右の2つの数字は80であって00ではない)．最後に得た値はSAVEに与えるための数字である．

CC.COMとCLINK.COMは，同じ構造の5バイトの設定ブロックを含んでいる．CC.COMのブロックの基本アドレスは0155hで，CLINKのブロックの基本アドレスは0103hである．ブロックの構造は次のようになる．

アドレス	機　　能	デフォルトの値
base+0	デフォルトのライブラリ・ディスク	FF (現在値)
base+1	デフォルトのライブラリ・ユーザー領域	FF (現在値)
base+2	SUBMITファイルが処理されるディスク	00(ディスクA)
base+3	割り込み(0か1)のためのポール・コンソール	01(可能)
base+4	処理終了後，ワームブートを行う	00(行なわれない)

ブロックの中の各項目は，1バイトの長さである．

最初の2バイトの設定値は，デフォルト・ディスクとユーザー領域を指定するもので，CCとCLINKではそれぞれ，“ライブラリ・ディスク”，“デフォルト領域”として扱われる．CCでは，ライブラリ・ディレクトリは#includeディレクティブ内の各角弧 < > で囲まれているファイルをみつける場所を補足し[6]，CC2.COMをみつける場所も指定する．CLINKでは，DEFF.CRL. DEFF2.CRL. DEFF3.CRL（もしあれば）とC.CCCを格納している場所を指定し，CLINKコマンド行に指定された“main”CRLファイルからディレクトリ内にない全てのCRLファイルの格納場所も指定する．

デフォルトのライブラリ・ディスクの値は，0がドライブAを指定し，1はドライブBを指定し，FFh (10進数で255) の値は，現在ログされたディスクがデフォルトのライブラリ・ディスクとして使用される．デフォルトのライブラリ・ユーザー領域のために，0－31の値は符号するユーザー領域を表わし，FFh

---

6)#include文で指定される二重引用符で囲まれたファイルネームは，設定とは無関係に指定されたファイルを求めて，現在のディレクトリを検索する．

(10進数で255)の値は現在のユーザー領域はデフォルトのユーザー領域になることを指定する．ライブラリ・ディスクとユーザー領域はFFhへ設定されているので，現在ログされたドライブとユーザー領域には常にライブラリのファイルを含んでいると仮定される．

3つめの設定値は，"Submit file"の処理中に存在する$$$.SUBファイルがどのドライブにあるかを指定する．可能な値は，前記のようにデフォルトのライブラリ・ディスクのためのものと同じである．

CLINKは，エラーが起こったときは，いつも保留中のサブミット・ファイルを消そうとする．一方，CCは，－xオプションが与えられたときにそうしようとするだけである．というのは，ほとんどのシステムは常にドライブA上に$$$.SUBファイルを置くからであり，これはCCとCLINKがデフォルト値によって設定されるようになる方法である．しかし，ユーザーが$$$.SUBファイルをいつものドライブAのかわりに，いわゆる現在のドライブ上に入れるためにシステムをカスタマイズするなら，そのときはこのバイトを，01hからFFhへ修正すればよい．

4つめの設定値は，コマンドの実行中に，システムのコンソールを割込みコード（コントロール－C）のためにポールするべきか，否かをCCあるいは，CLINKに告げるフラグである．もし可能になったら(ゼロ以外のもの)コマンドの実行中，コントロール－Cがタイプされなければユーザーによってコンソール上にタイプされた入力は無視される．コントロール－Cがタイプされた場合，処理はすぐに打ち切られ，コントロールはコマンド・レベルにもどる．もし，無効になったら（ゼロ）コンソールはポールされない．コンソールをポールするために，非常駐のコマンドを必要とすることもなく，割り込みを有するシステムではタイプアヘッドを利用することができる．

5つめの(そして最後の)バイトは，CCとCLINKが，それぞれの処理を終えた後，直接CCPに戻るか，あるいはワーム・ブートを行うかの指定を行うものである．デフォルトでは，CCPへ直接戻るように設定されているが，CP/Mライクなシステム（CROMIX上のCP/Mシミュレーターは1つの例であると聞いたことがある)上で，CCPへ直接的にもどるのは，正しく作動しない．こ

れはたぶん，OSが有効スタック・ポインタを非常駐コマンドへパスしないからであり，CCやCLINKが戻ろうとしたとき，システムをクラッシュしてしまう．もし，ユーザーがコンパイラを用い正しい出力ファイルを書き込んだ後に爆走する場合，ワーム・ブート・バイトをゼロ以外の値にセットする必要がある．

かいつまんでいえば，この設定案によって，大きな容量のディスクを持つユーザーがすべての標準ヘッダーと，ライブラリ・ファイルをキープする特殊なドライブとユーザー領域を指定することを可能にする．ライブラリ・ディスクとユーザー領域のバイトは，一つの単位として共に考えられるべきである．もしユーザーが一つの修正を行えば，たぶん他のものも変えたくなるだろう．

CC2. COMは，設定する必要はない．CC. COMは，適切な情報のすべてを含めてCC2.に処理を渡す．

1.4.2.2 ランタイムパッケージのオプション

CCC. ASMは，ユーザーのカスタマイズできるいくつかのEQUシンボルを含んでいる．ランタイムのパッケージを再びアセンブルしたり，ライブラリを適切に修正する方法についての詳しいことは第2章を参照のこと．

初心者のユーザーにとって，ランタイムパッケージの再設定のプロセスは多少やっかいなので，注意すること．

CCC. ASMの中のNFCBSシンボルは，開かれるファイルの最大数を規制する．これは供給されるバージョンでは8にセットされている．一度に開かれるファイルがもっと必要なら，単にこれを，必要な値（各々のファイル追加はランタイムパッケージが38バイトほど大きくなる）に変更すればよい．

1.4.2.3 BDSCIO. HとHARDWARE. Hのコンフィグレーション

標準I/OヘッダーファイルBDSCIO. Hは，NSECTSという定義された値をもっている．これはバッファ付きI/Oライブラリが，コントロールするバッファの大きさを定義するものである．NSECTSは8に設定されているから，ディスクへのアクセスがおこる前にバッファ付きI/Oオペレーションが，1024バ

イトのデータをバッファする．もし，ユーザーがCP/MのBIOS（基本・入力／出力システム）の中で，ブロッキング／デブロッキング化する1Kのセクタを持つシステムを実行中なら，そのときは冗長バッファリングを削除したり，オープン・ファイルごとに7/8Kバイトのフリーメモリーを得るために，ユーザーはNSECTSを8から1へ修正したくなるかもしれない．

I/Oポート・ナンバーやマスクなどのシステム従属ハードウェアの特徴は，HARDWARE.Hというヘッダーファイルの中で定義される．これを含むプログラムがコンパイルされる前に，HARDWARE.Hは，ターゲットコンピュータ・システムのハードウェアの特性を基にして修正されるべきである．

1.4.3 典型的なコンパイル作業

例として，ここにFOO.Cと呼ばれる単一のソース・ファイルをコンパイルしたり，リンクしたりする順序をあげる．

コンパイラは，次のコマンドで呼び出される

```
A⟩ cc foo.c ⟨Cr⟩
```

サイン・オン・メッセージをプリントした後，CCはディスクから，ファイルFOO.Cを読み取り，そして，中間コードを作成し始める．CCの処理が終了したら，メモリーの使用状態を表示し，CC2.COMをロードする．CC2では，CRLファイルを作成するために中間コードを解析し，エラーが起こらなければファイルFOO.CRLをディスクへ書き込む．

次のステップでリンカーが起動する．

```
A⟩ clink foo [他のファイルや必要なオプション] ⟨cr⟩
```

未決定の関数参照がなければ，ファイルFOO.COMがつくられ

```
A⟩ foo [引数] ⟨cr⟩
```

によって実行される．

重要事項： パッケージの中のすべてのCOMファイルのコマンド行は**先行ブランクなしで**CP/Mへ，タイプ・インされなければならない．これはコマンド行での先行ブランクがコンパイラによって，生成されたCOMファイル内のargcとargvの値の計算ミスを引き起こすからである．

つづいて，コマンドの文法の詳細である．

1.4.4 CCパーサ

コマンド・フォーマット：CC name. ext ［オプション］〈cr〉

Cプログラムの標準の拡張ファイルネームは，“.C”であるが，どんなネームも拡張ファイルネームも指定することが可能である．CCはまず指定されたファイルをオープンしようとする．拡張ファイルネームが指定されていなければ，また指定したファイルが，見つからなければ，CCはそのファイルネームに“C”という拡張ファイルネームを付加し，新しく設定されたネームでそれをもう一度オープンする．もし，ファイルネームがディスク名を含んでいるならば（例えば“b:foo. c”)ソースファイルは，そのドライブ上に存在すると仮定される．そして，コンパイラの出力もそのディスクに対して行なわれる．#includeディレクティブに対して，二重引用符の中で与えられたユーザー領域／ドライブの指定なしのファイルネームはコマンド行上に与えられたマスタ・ファイルネームと同じディスクから得られる．

CCがコンフィグレーションセクションで示されたような“コンソール入力のキャンセル”指定が**されていない**ときは，コントロール－Cをタイプすると，コンパイル処理は中断され，オペレーティングシステムに戻る．

ソースファイル名の指定の後に，コンパイル作業のオプションを指定することができる．オプションはマイナス記号で先行する．現在サポートされているオプションは，下記のとおりである．

－p は，すべての#defineと#includeディレクティブの展開が終了した後

に，行番号を付けて，ソース・テキストをユーザーのコンソール上に表示する．このオプションは，不一致のコメント・デリミタを探知するのに便利である．－pを指定すると，閉じられていないコメントは引き続くテキストすべてを消してしまう．また，最後の目に見えるテキストはまちがって区切られたコメントが始まっている場所をユーザーに教える．この出力は，CCを呼びだす前に，コントロールPをタイプすることによって，CP/Mの“LIST”装置へ出力される．

－a d[n]はCCの処理が正常に終了した後，ディスクdのユーザー領域nからCC2.COMを自動ロードする．デフォルトでは，CC2は現在ログ・インされているディスクか，設定手順で定義されたデフォルトのドライブ／ユーザー領域のところかのいずれかからロードするとみなされる．文字“Z”がドライブ名として与えられる場合，中間ファイル“.CCI”を後のCC2の呼び出しのために，ディスクへ書き込む．そしてCC2の自動ロードは行なわれない．

－d xは，CCかCC2の処理中にコンパイルエラーが起きなければ，コンパイラのCRL出力がディスクxへ書き込まれるようにする．－a zオプションが指定されている場合，－dは，.CCIファイルを書くディスク名を指定する．デフォルトのディスクはソース・ファイルが含まれているディスクと同じものである．

－m xxxxは，CP/M以外の環境でコードを生成するためのコンパイルのために，ランタイムパッケージ（C.CCC）のスタートする位置を16進で指定する．ランタイムパッケージは，デフォルトではCP/MのTPAのスタートから常駐する．もし，代替アドレスがこのオプションの使用によって与えられるなら，リンクする前にランタイムパッケージを与えられたロケーション用のマシーンコードとして，必らず再アセンブルすること．また，CLINKを使用するとき，適切なアドレス値を－l －e －tを用いて，与えることも忘れないように，BDS CのオブジェクトをCP/M以外の環境にカスタマイズすることに

関する詳しいことは，第2章をみよ．生成されたCOMファイルの先頭に位置するC.CCCは，mainと，他のルート・セグメント関数（もしあれば）を，分離することができない．－mを指定した場合は，CCによってCC2が自動的にロードされなければならない．

－e xxxxは，ランタイムの外部データ領域のスタート・アドレス（16進）の指定を行う．普通，外部データ領域は，プログラム・コードの最後のバイトのすぐ後に続けて始まる．そして，ランタイムパッケージの，すべての外部データはCLINKによって設定される間接的手段によるポインタによって，アクセスされる．－eが使用されるとき，外部データ・ポインタを使用するかわりに，コンパイラは外部データを直接的にアクセスするための（lhldとshld命令を使用して）コードを生成することができる．これは多くの外部データをもったプログラムのパフォーマンスを高めることになる．ただしデバック中のプログラムは，このオプションを使用しないこと．プログラムがうまく動作したなら－eと前述したような外部データ領域の値を指定して再びコンパイルする．（前回よりも，オブジェクトコードは，短かくなる．）コードサイズがどれだけ縮められたかを見つけ出すために，CLINKは“最後のコード・アドレス”の値を表示する．それから，－eオプションで適切な低い方のアドレスを指定して，再びそれを全部コンパイルする．しかし，あまり近づけすぎない方が良い．というのは，ユーザーはプログラムの変更を行なえばそれに対してサイズは変動するようになるので，おそらく，指定された外部データ領域はオーバーラップ（CLINKが検出し，レポートする）することになる．－eを指定した場合は，CCによってCC2が自動的にロードされなければならない．CLINKオプション－eにも，関連する事項があるので参考にすること．もし，このオプションを指定した場合，外部データアドレスが最後のコマンド・ファイルの中のプログラムやOSのある部分とオーバーラップするなら，CLINKは警告メッセージをプリントする．

－oを指定すると，実行速度に対して最適化されるようなコードを生成する．普通，ランタイム・パッケージ内の単調なコードシーケンスを，共通のサブル

ーチンと置換する．これはコードの長さを小さくするが，サブルーチンへのリンクのオーバーヘッドのために，実行速度がスローダウンする結果になる．－o が使用される場合，それらのサブルーチン・コードの多くは，イン・ラインコードに展開されるので，より速い（しかしより長い）オブジェクト・プログラムが生成される．最も高速なコードを生成するには，－e オプションと－o とを，同時に指定する．できるだけ短いコードのためには，－e だけを使用し，－o を使用してはいけない．－o を指定した場合は，CC によって CC2 が自動的にロードされなければならない．

－x を指定すると，コンパイル作業中にエラーが発生したら，CP/M "SUBMIT" ファイルを削除する．CC が SUBMIT ファイルから使用されるときは，コマンド行に－x スイッチを指定すべきであり，そうすると，コンパイルエラーの発生にともなって，コマンド・レベルに戻るまえに，"$$$. SUB" の一時的ファイルを削除する．CC が単独で使用されるとき，－x は必要のないディスク動作を引き起こすので，使用するべきでない．

－r x は，x K バイトをシンボル・テーブルのために，用意することを意味する． "Out of Symbol space" のエラーが起こったなら，このオプションは，シンボル・テーブルのために割り当てられるメモリー空間の量を増やすために使用される．もし，ユーザーが "Out of Memory" エラーを引き起こしたなら，－r は，シンボル・テーブル・サイズを減じるために使用され，ソース・テキストのためにもっと多くのメモリー空間を与える．けれど，"Out of Memory" が発生した場合のより良い解決手段は，ソースファイルをより小さなモジュールに分割することである．デフォルトのシンボル・テーブル・サイズは，10K である．

－c は，"コメントの入れ子" 機能を無効にし，コメントが UNIX C によるのと同じ方法で処理されるようにする．たとえば，－c が与えられたとき，

```
/* print f(" hello "); /* this prints hello */
```

のようなラインは，全てコメントとみなされる．－cが使用されない場合，コンパイラは，こういったコメントが終結したとみなさずに，次の*/が現われるまでコメントとする．

ひとつのソースファイルは，63以上の関数定義を持つことができない．とはいうものの，Cプログラムはいくつものソースファイルからつくりあげられるし，その各々は，63までの関数を持つことができるのである．

CCによってエラーが検出されたなら，コンパイル作業は中断され，二番目のコンパイル作業のフェーズ(CC2)へと進んだり，CCIファイルをディスクへ書き込む（どのオプションが与えられたかによって）ことはない．

実行スピード： 約20行／秒．ソースファイルがメモリーへロードされた後には，ディスク・アクセスは，コンパイル処理が終わるまで，発生しない．最後のディスク動作が起こってから，少なくとも（n／20）秒が経過するまでクラッシュが起こったと想定しないこと(それ以後は想定した方が良い)．ここで，nというのはソース・ファイルの行数である．

例：

```
A> cc foobar.c -r12 -ab <cr>
```

CCを呼び出し，ファイルfoobar.cをコンパイルする．シンボル・テーブル・サイズは，12Kバイトにセットされ，CC2.COMはディスクBから自動ロードされる：

```
A> cc c:belle.c -p -o <cr>
```

CCを呼び出し，ディスクCからファイルbelle.Cをロードしコンパイルする．テキストは，#defineや#Includeの処理の後，コンソール上に（行番号を付けて)出力される．CCがエラーをみつけなければ，CC2.COMは現在ログされた

ディスクかデフォルトのドライブ／ユーザー領域(セクション1.9.2で設定されている）のいずれかから，自動ロードされる．そして，オブジェクト・コードはスピードアップのために最適化される．

1.4.5 CC2 コードジェネレータ

コマンド フォーマット： CC2 name <cr>

普通，CC2.COM は，CC によって自動的にロードされるので，このコマンドは使用される必要はない．コマンドが与えられた場合，ファイル name.CCI は，メモリーへロードされ，処理される．

エラーがなければ，name.CRL と呼ばれる出力ファイルが生成され，name.CCI（もしあれば）は削除される．

CC2 にはオプションはない．

CC では，ファイルネーム内にディスク名を指定すると，そのディスクが入力と出力のために使用されるようになる．

CC が CC2 を自動ロードするとき，CC2 内のいくつかのバイトは，CC のコマンド行に与えられたオプションに従って，セットされる．CC2 が単独で呼び出される場合（CC によって自動ロードされないとき）CC2 が実行を始める前に，ユーザーはこれらの値が指定された値にセットされていることに気をつけなければならない．これは必要なことではないが，もし，ユーザーが容量の小さなディスク装置から CC2 を別に呼び出す必要があるなら，セットされる必要のあるデータ値は次のようなものである：

アドレス	デフォルト	オプション	機 能
0103	00	−a	CC2が自動ロードされたなら，真，他はゼロ
0104	01	−o	−oが与えられたなら(実行速度の最適化)ゼロ，他は1
0105-6	0100h	−m	ランタイム・オブジェクトのC.CCCの開始アドレス
0107-8	none	−e	外部データ領域の開始アドレス (CCに-eが与えられたなら)．
0109	00	−e	外部データ領域の開始アドレスが与えてくれる場合真，他は0

16ビットの値は，上下バイトを逆にすること．(最初は，低い方のバイト，次は高い方)

CC2の実行スピード： 約70行／秒 (元のソース・テキストに基づく)

実行中に，コントロール−Cが入力されたら，コンパイル作業は打ち切られコントロールはコマンド・レベルに戻る．

例：

A> CC2 foobar <cr>

1.4.6 CLINKリンカー

コマンド フォーマット： CLINK name [他のファイルネームまたはオプション] <cr>

ファイルname.CRLは，main関数をもっていなければならない．name.CRLと，指定された他のすべてのCRL (−f) オプションが出てくるまで) ファイル内の関数はすべてメモリーにロードされ，リンクされる．−fオプションがコマンド行に現われたなら，それに続いて指定された**すべての**CRLファイルは，必要な関数を**走査**するためのものである．それは，(前のCRLファイル

か，現在走査されているファイルのいずれかから）前もってロードされた関数によって必要になるとわかっているもののみをロードし，リンクする．指定されたCRLファイルすべてが，検索された後，標準ライブラリ・ファイルDEFF＊.CRLを必要とされるライブラリ関数のために自動的に走査する．ライブラリ・ファイルが検索される順序は，いつも同じである．最初は，DEFF.CRL，次にDEFF2.CRL.そして最後に，ユーザーが供給するなら，DEFF3.CRL.もし，ユーザーがこれらの自動検索されるライブラリ・ファイルの中の関数と同じ名前を持つ関数を書いたなら，こういった関数は常に，コマンド行に指定されたCRLファイルの1つの中におかれるべきである．DEFF3.CRLの中におかれる場合，DEFF.CRLとDEFF2.CRLの中で同様に指定された関数が，削除されない限り，読み込まれることはない．

CLINKはデフォルトで，すべての指定されたCRLファイルがディスク上にあると仮定し，すべてのライブラリ・ファイル（C.CCCとDEFF＊.CRL）はコンフィグレーションのセクション（セクション1.4.2をみよ．）の中で定義されたようなデフォルトのドライブとユーザー領域からそれぞれロードされる．ドライブ名をコマンド行のmain関数を含んだファイルのファイルネームの前に置くなら，与えられたドライブは，コマンド行で指定されるCRLファイルすべてのためのデフォルト値になる．各々の追加CRLファイルは“d：”というフォームのディスク指示子と，“nn/”というフォームのユーザー領域の指示子もしくは，その一方をファイルをさがす明白な場所を指定するために含んでもよい．両方の指示子が使用される場合，ユーザー領域の指示子をはじめに指定しなければならない．

指定されたCRLファイルが上記の検索ルールに従ってみつけられない場合，デフォルトのドライブとユーザー領域(セクション1.4.2をみよ)によって，指定されたディレクトリも検索される．これは，ユーザーが1つのデフォルトのドライブ／ユーザー領域の中に，共通に使用されるライブラリ・ファイルをおくことを可能にし，リンク作業に異なったドライブとユーザー領域をアクセスできるようにする．

与えられたすべてのCRLファイルが検索された後，未定義の参照が残って

いると，CLINKは“会話型モード”に入る．ここでCLINKは未定義の関数名を表示し，他のCRLファイルを検索することを可能にする．

実行中にコントロール－Cがタイプされると，リンク処理は中断され，コマンド・レベルへ戻る．

リンク作業のオプションは，ファイル名との混合をさけるためにオプションの前にマイナス記号を付ける．1つのマイナス記号の後に，いくつかのオプションを，一度に指定してもよい．現在使用できるオプションは，下記のとおりである．

－s コンソール上にロード・マップとモジュールのサマリを出力する．

－f（ファイルネーム…）は，続けて指定されたCRLファイルのすべてを**ロード**するかわりに，**走査**されるようにする．CLINKはこのオプションに出会うまで，コマンド・ライン上で指定された各々のCRLファイルの中のすべての関数をロードする．－fオプション以後に続くCRLファイルはロードされずに走査される．これが意味しているのは，はじめのころの（いくつかのファイルや現在のファイルの中で他の関数によって，前もって参照された関数だけが，プログラムにリンクされるということである．

注意： この新しい－fオプションは，BDS Cの1.50バージョン以前の－fとはちがった動作をする．－fはL2リンカーの“－L”オプションと同様に動作する．

－e xxxx は，外部データ領域の開始アドレスをxxxx（16進）で指定する．普通，外部エリアは生成されたコードのすぐ後から開始されるが，－eオプションを指定すれば，この機能は無効になる．このオプションは，コマンドをチェインする場合，新しいコマンドの外部データ領域の開始アドレスを，前に実行したコマンドと同じアドレスにしたい時には必要である．この値を見出すためにはまず，最初に全てのCRLファイルと－sオプションを用いて，CLINKを実行する．－sオプションなしでは，CLINKは，それぞれのファイル

の"External Start At:"の表示をしない．それから前述の－sオプションで得られた**一番の大きな**"External Start at"アドレスを，－eオプションのオペランドとして与え再度リンクする．ROMのためにコードを生成するときは，このオプションは，RAMの適切な位置に外部データ領域を置くために使用されるべきである．main関数を含んだCRLファイル（name.CRL）が，CCの－eオプションを用いて，コンパイルされたなら，CLINKはCCコマンド行で指定されたアドレスを自動的に検知する．しかし，もしリンク作業で指定された他のCRLファイルのどれかが，－eオプションを使用しなかったり，main関数とは異なる－eの値を使用して，関数をコンパイルした場合には，出てくるCOMファイルは正しく作動しない．CCの－eオプションの使用なしでコンパイルされたCRLファイルは，main関数を－eを使用してコンパイルしたときのアドレスと全く等しい値をCLINKの－eオプションで指定した場合に限って，正しくリンクされる．

－zは，ランタイムの初期化のとき，外部データ領域をゼロにクリアするのを抑止する．－zが使用される場合，すべての外部のものはランダムな値になる．さもなければ，外部のものは全てゼロにクリアされる．

－t　xxxxは，予約されたメモリーのスタートをxxxx（16進で与えられる）にセットする．命令lxi sp, xxxxが生成されたCOMファイル[7]の最初に置かれる．CP/Mではランタイム内の全てのプログラムコードと内部／外部データエリアの大きさが，指定された値に入りきるような十分な大きさでなければならない．もし，ROMベースで走らせるプログラムを作成するなら，この値は，RAMの最終アドレス＋1に設定しなければならない．

7）普通－tが使用されないとき，生成されたCOMファイルは次の順で始まる．

```
lhld bose+6    ; ベースページからBDOSポインタを得る．
sphl           ; BDOSのベースへスタック・ポインタを設定する．
```

－o newname は，COM ファイルの出力が，newname. COM と名付けられるようにする．ディスク名がその名前に先立つなら，その出力は指定されたディスクへ書き込まれる．デフォルトの出力は現在ログされたディスクである．コロンと，その前に置かれたディスク名を，ファイルネームなしで与えるなら，COM ファイルは，ファイルの名前を変更することなく，指定されたディスクへ書き込む．

－n は，ランタイム・スタックが CP/M の CCP（コマンド・プロセッサ）とオーバーレイするかわりに，CCP を保存させるような値に設定し COM ファイルを出力する．これはランタイムのメモリーを 2 K バイトほど減じるが，実行終了後，**ディスクからワーム・ブートする必要なしに**コマンド・レベルへ戻ることができる．したがって，－n は頻繁に使用されるプログラムやメモリーをちょっとしか必要としないプログラムにとって便利である．このオプションは，NOBOOT コマンドを実行するのと同じような影響をもっている．NOBOOT は，L2 のような他のリンカーとリンクされたプログラムが，ワーム・ブートを行なわずに CCP へ戻るようにするために，与えられている．－n は，－t オプションが同時に指定されているときは無視される．なぜなら，そのメカニズムは相入れないもので，－t が優先権を与えられているからである．

－w は name. SYM というシンボル・テーブル・ファイルをディスクへ書き込む．ここで name は生成された COM ファイルのものと同じである．このシンボルファイルは，シンボル名とリンクに含まれたすべての関数の絶対アドレスの値をもっている．これは，デバッグする目的のため，SID を用いて使用され，あるいはオーバーレイ・セグメントをつくる時，－y オプションによって使用される（下記をみよ）

－y sname は，ディスクから指定されたシンボル・ファイル sname. SYM を読み取る．そして，現在のリンクのためそこで定義された関数の名前すべてのアドレスを使用する．－w と－y オプションは，オーバーレイをつくるために

一緒に作動するように，次のように設計されている：'ルート'セグメント（TPAからロードされ，最初にコントロールを受けるランタイム・ユーティリティ・パッケージを含んでいるプログラムの部分）をリンクするとき，－wオプションは，ルートの中で現われるすべての関数のアドレスを含んでいるシンボル・テーブル・ファイルを書き出すために与えられるべきである．それから，オーバーレイ・セグメントをリンクするとき，－yオプションは"親"ルート・セグメントのシンボル・テーブルの内容を読むために，使用されるべきである．そして，それによって，一般的なライブラリ関数の多重コピーが，ランタイムにあらわれるのを防ぐ．この手順は，もう１つ下のレベルにも用いることができる：オーバーレイ・セグメントをリンクするとき，－wオプションは，－yオプションと共に与えることが可能であり，それを行うと，新しく定義される局所関数と，読み込んだシンボルの定義を含んだシンボルファイルが生成され，この操作は，必要なだけレベルを下げることができる．CLINKコマンド行の－yオプションの位置は，重要である．つまり，－yオプションで指定されたシンボル・ファイルは，－yオプションの左に指定されたCRLファイルの検索が終った後にだけ，検索される．このように最長の結果はmain CRLファイルネームのすぐ後に，－yオプションを指定することである．もしSYMファイル内のシンボルがすでに定義されたシンボルと同じ名前であるなら，その影響に対するメッセージがコンソールに出力され，そのシンボルの古い値が保持される．オーバーレイ・セグメントを生成するための－yの使用に関しては，"付録D"をみよ．

－l xxxx 生成されたコードのロード・アドレスが，xxxx（16進）になるようにする．このオプションは，オーバーレイ・セグメント（－vと関連して）を生成するとき，あるいは，標準外の環境で実行するコードをつくるときに，必要なだけである．後者の場合，CCC.ASMは適切なアドレスのために再設定されてなければならないし，xxxxを用いているC.CCCの新しいバージョンをつくるために，再びアセンブルしなければならない．この場合，－eと－tオプションもまた，適切なRAM領域を指定するため，使用されるべきである．

－vは，オーバーレイ・セグメントを作成することを指定する．作成されたコードの中には，ランタイム・パッケージが含まれない．というのは，ランタイムのパッケージのコピーがデフォルトによるTPAのベース，あるいはCCの－mオプションの中で指定されたアドレスのいずれかで，すでに常駐していると想定されるからである．

－cd［n］は，**ライブラリ・ファイル**（DEFF.CRL. DEFF2.CRL. C.CCCと，もしあればDEFF3.CRL）と，コマンド行で指定されてはいるが，現在のドライブ／ユーザー領域でみつからないCRLファイルを，ディスクdとユーザー領域nから，取り込むようにCLINKに命令する．このオプションは，CLINKコンフィグレーション・セクションで設定されたデフォルトのドライブ／ユーザー領域の仕様を無効にするために使用される（セクション1.9.2をみよ）

－d［"args"］－dは，すばやくテストするために，COMファイルとしてディスクへ書き込むかわりに，リンク作業によってつくられたCOMファイルがすぐに実行されるようにする．引数のリストが二重引用符で囲まれて指定されているならそれは，CCPからのパラメータと同じものになる．

－dは，TPAのベースのところとはちがうロード・アドレスを持つセグメントのために使用することはできない（つまり，－dは，ルート・セグメントのために使用される）．内部の矛盾のために，－nが与えられている場合は，－dは無視される．

－r xxxxは前方参照のテーブルのためにxxxx（16進）バイトを予約する．（デフォルト値は約600h）"Ref table overflow"のエラーが起こったときは，このオプションはもっと多くのテーブル・スペースを割り当てるために使用される．

例：

A〉 clink ted −s −t 6000 −o joyce 〈cr〉

ここで，CLINK はファイル TED. CRL が main 関数を含んでいると仮定する．TED. CRL のすべての関数と，DEFF. CRL. DEFF2. CRL そして存在するなら DEFF3. CRL[8)]から必要な関数とがリンクされる．リンク終了後モジュールのサマリがコンソールに出力される．実行時のスタックは 6000h からスタートし，下位アドレスに向って増えていく(6000h 以後のメモリーは COM ファイルによって手をつけられない)，また COM ファイル自身は JOYCE. COM という名前になる．

A〉 clink b:lois 6/c:vicky −f janet −s 〈cr〉

この例で，CLINK は（ドライブB：の）LOIS. CRL と（ドライブC：のユーザー領域 6 の）VICKY. CRL からすべての関数をロードし，JANET. CRL（ディスクBから，というのは LOIS. CRL が入っているディスクがこのリンク作業でデフォルトになる）と DEFF. CRL:それからたぶん DEFF3. CRL（セクション1.9.2によって設定されたデフォルトのディスク／ユーザー領域から)から必要な関数をリンクする．そして，リンク終了後，コンソールにモジュールのサマリを出力する．−t オプションは与えられないので，CLINK はすべての TPA（非常駐プログラム領域）が，コードとデータにとって有効であると想定している．生成された COM ファイルはデフォルトによって LOIS. COM と指定される(−o オプションは与えられてなかったから)．そして，ファイルは現在ログ・インされたディスクに書き込まれる．

注意： 外部の変数を共用するいくつかのファイルが，共にリンクされるとき main 関数を含んでいるファイルは，他の全部のファイル

8) DEFF3. CRL がユーザーの供給したライブラリ・ファイルとしてもし存在するならば，DEFF. CRL と DEFF2. CRL が走査されてしまった後にまだ未定義のシンボルがあると，自動的に走査される．DEFF3. CRL が存在しない場合，リンカーは何もしない．

の中で使用される外部変数の**すべて**の宣言を含んでいなければならない．これは，リンカーがmainの含まれているファイルから得た外部領域のサイズを用いるからであり，この値はライブラリ関数のendext ( )が正しい値を戻せるように，また出てきたCOMファイルの中に適切なパラメタをセットするために，使用されるのである．また，BDS Cの外部変数は，UNIX Cの外部変数のものよりもっとFORTRANのCOMMON文に近いので，外部宣言の配列は，プログラムの個々のソース・ファイルの範囲内では等しくあるべきである．このように，プログラムの各ファイルに含まれているヘッダーファイルには，互換性を確実にするためにすべての外部宣言を含める．

1.4.7 CLIB-C ライブラリ管理プログラム

コマンドフォーマット： CLIB 〈cr〉

CLIBプログラムは，ユーザーに次のような目的で供給される．

a）CRLファイル間で関数を転送させる

b）個々の関数を改名し，削除し，検査させる

c）新しいCRLファイルをつくらせる

d）CRLファイルの内容を検査させる

CLIB操作を始める前に，CRL（C再配置可能ファイル）ファイルの構造を理解しておくと役に立つ：

CRLファイルは，独立してコンパイルされたC関数の集まりで，0000の基点をもつ2進の8080マシーン・コードのイメージである．各関数と共に記憶されるものはCLINKによる使用のために再配置可能なアドレスを解くための“再配置パラメータ”のリストと，与えられた関数によってコールされた付随関数の名前である．コード，再配置リスト，必要関数リストの集合は1つの関数モジュールと呼ばれる．

CRLファイルの最初の4つのセクタは，そのファイルのためのディレクトリ

であり，ファイルに含まれる関数モジュールやそれらの位置情報を含んでいるCRLファイルのトータル・サイズは64Kバイトを超えることはできない（関数モジュールは2バイトのアドレスによって配置されるからである）．しかし，最も効率的なのはCRLファイルのサイズをCP/M・ファイルエクステント（16Kバイト）の大きさに，制限することである．

CRLファイルについての詳しいことは，第3章をみよ．

CLIBが起動されると，タイトルメッセージと“関数バッファ・サイズ”を表示する．バッファ・サイズは，転送処理の間にどれくらいのメモリーが関数の中間記憶に用いられるかを示す．大きな関数を転送したり，引き出したりすると，動作しないことがある．

初期化に続いて，CLIBは，アスタリスク(*)をプロントし，コマンドを待つ．

CRLファイルを処理するには，まず，CRLファイルを次のようにオープンする．

* open file # [d:] filename ⟨cr⟩

ここで，file #は，そのファイルがオープンされたままでいる限り，そのファイルのfilenameと関連されるための“ファイルナンバー”を指定する一桁の数の識別子（0－9）である．ゆえに，10までのファイルを同時に**オープンすることができる**．

ディスク名は，CLIBがどんな物理ディスク上からでも，CRLファイルを操作できるようにすることを可能にする．

ファイルをクローズするためには（ファイルに対してなされた**変更**を永続的にして），

* close file # ⟨cr⟩

与えられたファイル・ナンバーは，後のopen処理によって，新しいファイルへ割り当てられるようにするため，空になる．変更されたファイルのバックアップファイルは，名前，name.BRLを用いて作られる．close操作は，いく分

時間をとり，大きなファイルが呼ばれるときには，ユーザーのディスク・ドライブのわずらわしい騒音をひきおこすであろう．

ファイルに対して変更が行われない限り，あるいはユーザーが余分なファイル・ナンバーを必要としない限り，ファイルをクローズする必要はない．例えばコピーされるためだけにオープンされたファイルはクローズする必要はない．

CRL ファイルがオープンされると，ファイルのディレクトリ（最初の4セクタ）のコピーがメモリーへロードされる．ファイルに対して変更が行われると（append，transfer，rename，delete のコマンドによって）メモリーの中にあるディレクトリもそれに順じて変更される．しかし，更新されたディレクトリをディスクへ書き戻すには，ファイルをクローズしなければならない．このように，もしユーザーが，今まで行なったことを，すべて取り消したい場合は，単に，そのファイルをクローズしなければよい．

append，transfer の処理を取り消したい場合は，少し余分な作業を必要とする．これは後で説明する．

すべてのオープンファイルの関連統計のリストを見るには

```
* files 〈cr〉
```

とする．

具体的な CRL ファイルの内容をリストし，その中の各関数の長さを知るためには，次のコマンドを用いる．

```
* list file # 〈cr〉
```

CRL ファイル間で関数をあちこち動かすにはいくつかの方法がある．関連ファイルすべてがオープンされたとき，関数をコピーするのに一番の近道は，

```
* transfer source-file # destination-file # function-name 〈cr〉
```

これは，指定された関数をソース・ファイル source-file #から，目的ファイル destination-file #にコピーし，ソース・ファイルのオリジナルは，削除しな

い．function-name に曖昧な指定をするには，＊と？の文字を使用する．このときは，曖昧な名前と一致するすべての関数が転送される．

ファイルをあちこち動かすための他の方法は，“extract-append”の方法を使用することである．

extract コマンドは次のフォームをしている．

* extract file # function-name 〈cr〉

これは，与えられたファイルから単一の関数を引き出すために，また，関数バッファ（メモリー）の中にそれをストアするのに使用される．その関数をファイルに書き出すためには，

* append file # [name] 〈cr〉

とする．ここで，name はオプションであり，関数を書き出すときに，関数の名前を指定する．

* append file # 〈cr〉

は，関数の名前を変更せずに，その関数をファイルへ書き出すために用いられる．append は，1度に1つしか file #を指定できないので,関数を,いくつかのCRL ファイルに書き出すには，書き出すファイルごとに append を使用しなければならない．

CRL ファイル内の関数の名前を改名（リネーム）するためには，

* rename file # old-name new-name 〈cr〉

これは，そのファイルに対して変更されるので，close はその変更を永続的にするために行なわなければならない．

新しい（空の）CRL ファイルをつくるには，

* make filename 〈cr〉

とする．これは filename. crl と呼ばれるディスク・ファイルをつくり，そのデ

ィレクトリを空に初期化する．それに対して関数を書き込むためには，最初に open を使用し，それから transfer か extract/append といった方法を使用する．CLIB は，同じディレクトリの中に存在する CRL ファイルと同じ名前を持つ新しい CRL ファイルを作ることを認めない．

ファイルから，1つの関数を（あるいは，関数の集合を）削除するためには，

 * delete file # function-name <cr>

を使用する．

関数の名前は，＊と？といった文字を使用して，曖昧な指定をすることもできる．ファイルに対する関数の削除を終結するためには，続いてクローズする必要がある．関数を削除しても，そのファイルが実際にクローズされるまでは，関連する CRL ファイルの中のどんなディレクトリスペースをも解放しない．このように，CRL ファイル・ディレクトリが一杯になっていて，その関数のいくつかを置き換えたい場合，まずはじめに，不必要な関数を削除し，それから，新しい関数をファイルの中に転送するために，いったんクローズし再びオープンしなければならない．

コマンド構文要約は，次のコマンドをタイプすることによって，参照できる．

 * help <cr>

CLIB を終了して，コマンド・レベルへ戻るために，次のコマンド

 * quit <cr>

を与えて，CLIB が表示する確認メッセージに，必ず答える．

すべての CLIB コマンドは，最初の一文字だけで指定することができる．

万一，ファイルに対する変更を，本当はしたくなかったのだと決めたなら，そして，それがすでにすんだことであったなら，quit コマンドはファイルの編集を終了するために，また，行われた変更をすべて中止するために使用される．ユーザーがファイルに付加したり，転送したりしない限り，

```
* quit file #  <cr>
```

とタイプすれば，そのファイル上の操作全部を打ち切って，close がなされたかのようにその file #を解放する．

もし，ファイルへのアペンドや転送がなされてしまってから，打ち切りたい場合も quit コマンドが使用される．しかし加えて，quit した後そのファイルを，再オープンし，それからすぐにファイルをクローズしなければならない．この手順の原理は次のようになっている：　ユーザーがアペンドや転送を行うとき，アペンドされている関数は CRL ファイルの最後のところから書き込まれる．また，ユーザーがそのエディットを打ち切るとき，古いディレクトリはもとのままにされる．しかし，アペンドされた関数はそのディレクトリの中に現われなくても，データ領域の中に依然として存在している．ファイルをオープンし，すぐにクローズすることによって，ディレクトリの中に現われる関数だけが，そのファイルと共に残り，結果的に他の“架空”の関数を取り除く．

次のサンプル・セッションは，DEFF. CRL ファイル内の頭文字が“P”で始まる関数すべてを，ドライブBに新しく作成する CRL ファイル NEW. CRL に転送するものである．

```
A > clib
BD Software C Librarian V1.50
Function buffer SIZE=xxxxx bytes

* open 0 deff
* make b:new
* open 1 b:new
* transfer 0 1 p *
* close 1
* quit
 (Quit)   Are you sure ? y
A >
```

1.5 CP/M の " Submit " ファイル

CP/M " Submit " ファイルは，Cプログラムをコンパイルしたり，リンクしたりするプロセスを簡単にするために作成されうる．初期のバグが除去されていて，かつ，ソース・ファイルの中にどんなエラーも存在しないと信用してから，自分でもっているCソースプログラムを単にコンパイルしたり，リンクしたり，実行したりするために，サブミット・ファイルの最も簡単なフォーム(関数をリンクするために他の特別な CRL ファイルを必要としない)は，次のように現われる．

```
CC $1.C
clink $1 -S
$1
```

ここで，LIFE.C と呼ばれるソース・ファイルをコンパイルしたい場合，ユーザーは，

```
A > Submit c life <cr>
```

とタイプすればよい（サブミット・ファイルは，C.SUB と命名されていると仮定する）．

1.6 新しい事項

1. BDS Cパッケージの中の COM ファイルや，コンパイラによって生成された COM ファイルを呼び出すとき，ユーザーのコマンド行（CP/M へタイプされるように）は，先行ブランクやタブを含んでいてはいけない．もし，先行するホワイト・スペースがある場合，CCP（コンソールコマンド・プロセッサー）は，正しい方法でコマンド行を展開しないようである．

2. MP/M IIで実行する場合，ランタイムパッケージ（CCC.ASM→C.CCC）の"MPM2"の定義を1にして再アセンブルしなければならない．これは，Cプログラムが実行している間，ランタイムパッケージが実際にオープンされたすべてのファイルをクローズするのを確実に行うためで，そのシステムはファイル・スロットを使い果たさない．普通，MPM2システム以外のもとでは，BDS Cランタイムパッケージは，読み取りのためだけにオープンされたファイルをわざわざクローズしない．

1.7 まとめ

バグなどが見つかった場合は，次の所までお知らせ下さい．

Leor Zolman
BD Software
P.O. Box 9
Brighton, Massachusetts, 02135
(617) 782-0836（夜の方が結構，東部標準時（EST）1：00AM（日本時間で2：00PM）まで）

技術的なバグのレポートで，業者の方と言い争うようなことはなさらないで下さい．バグレポートを直接BDソフトウェアに教えていただくことによって，かつて聞かれたことのある問題を知ることもできるし，ほんの少しの時間で，その欠点を補うことができるからです．

このコンパイラの開発のデバッギングの段階で，非常に貴重なフィードバックと御支援をいただいた次の方々に心より感謝申し上げます．

Lauren Weinstein
Leo Kenen
Rick Clemenzi
Sid Maxwell
Bob Mathias
Bob Radcliffe

Tom Bell
Jon Sieber
Scott Layson
Tony Gold
Ed Ziemba
Scott Guthery
Earl T. Cohen
Sam Lipson
Dan MacLean
Mike Bentley
Carlos Christensen
Perry Hutchinson
Paul Gans
John Nall
Mark Miller
Jason Linhart
Calvin Teague
Bob Shapiro
Cal Thixton
Jeff Prothero
The Real Cat
Al Mok
Phillip Apley
Charles F. Douds
Robert Ward
Les Hancock
Ted Nelson
Ward Christensen
Jerry Pournelle
Will Colley
Richard Greenlaw
Tim Pugh
Steve Ward
Tom Gibson
Roger Gregory
Don Lucas
Rev. Stephen L. de Plater
Nigel Harrison
Gary Kildall

特に，Dennis M. Ritchie 氏，Ken Thompson 氏，それから，UNIX やオリジナルのCの開発にあたったベル研究所のコンピューティング・サイエンス・リサーチセンターのスタッフの皆様方に心より感謝申し上げます．

BDS C のユーザーズグループは，しだいに組織化されてきました．このグループからの援助，ニュースレターやアップデートされたコンパイラの安価な入手方法についての連絡先は，

BDS C User's Group

P.O. Box 287

Yates Center, Kansas 66783

(316) 625-3554

なお，現在のところ，BDS Cパッケージははじめての方でもユーザーグループから購入できます．

第 2 章

CRL 関数フォーマットと低レベル関数

2.1 はじめに

本章は，コンパイルされたCの関数と，マシーン・コードのサブルーチンをリンクする必要のあるアセンブリ／マシン言語のプログラマーに対して向けられているものである．また，CRL フォーマットを詳しく説明しており，アセンブリ言語から CRL ファイルを作成し，Cリンカーによって他の関数のように取り扱われるようにするため，それらの関数をどうやって生成するかについても詳しく説明している．C関数のために使用されるパラメタを受け渡したり，コールしたりする規制も，ランタイムパッケージの中に存在する有用なサブルーチンと共に説明されている．

2.2 CRL フォーマット

BDS C のディスクに含まれている CASM.C は，CP/M の ASM アセンブラと共に使用されるためのものである．このプログラムによって，アセンブリ言語の関数を特別な " CSM " フォーマットで（1.46以前のリリースの " CMAC. LIB " マクロ・パッケージよりも，標準アセンブリ言語に近い．）記述することができる．それから，ASM. COM アセンブラのために，CSM ソース・ファイ

ルを ASM ソース・ファイルへと自動的に変換する．CASM.SUB と呼ばれる CP/M の"Submit"ファイルは，この手順のほとんどを自動的にするために与えられる．

CRL ファイルを生成する CASM と ASM を効果的に使用するために，CRL ファイルがどのようにして編成されるのかを知ることは，絶対必要という訳ではないが，CRL フォーマットの詳しい説明は，完全を求めてのことである．

2.2.1 CRL のディレクトリ

CRL ファイルの最初の 4 つのセクター[9]は，CRL ディレクトリと呼ばれる．そのファイルの中の各関数のモジュールは，ディレクトリの中に符号するエントリをもっており，モジュールの名前（8 文字までで，最後の文字は高位ビットが 1 になっている）と，そのファイル内[10]のモジュールのバイト・アドレスを示す 2 バイトの値から成る．

続いて，最後のエントリは，空白バイト（0x80）とそれに続く 1 ワード（2 バイト）のファイルの中の，次の有効アドレスを示すアドレスが続けられる．実際の関数モジュールの最後に，次のモジュールを偶数（16バイト）のアドレスから開始させるために，パッドが入れられることがある．ディレクトリの中には，どんなパッドも存在しない．

例： CRL ファイルが次のモジュールを含んでいる場合，

名 前	**長 さ**
foo	0 x 137
yipe	0 x 2c5
blod	0 x 94A

9）ファイルを調べるために，DDT や SID を使用している場合，これらのセクターはメモリー・ロケーション 0100h-02FFh にロードされる．

10）CRL ファイルのすべての関数モジュールの開始アドレスは，0x0000 から 0x01FF までに常駐しているディレクトリと共に 0x0000 を起点とする．ロケーション 0x200-0x204 は，リザーブされているので，一番低い関数モジュールの開始アドレスは 0x205 である．

そのファイルのためのディレクトリは，次のよう[11]に現われる．

46 4F CF 05 02 59 49 50 45 C5 50 03
F 0 0′ nn nn Y I P E E′ nn nn

42 4C 4F C4 20 06 80 70 0F
B L O D′ nn nn null-entry

2.2.2 外部データ領域の基点と大きさ

CRL ファイルの5番目のセクタの最初の5バイト(そのファイルの基点から数えて，0x200～0x204)は，CLINK が使用するランタイムの外部データ領域の基点(－e オプションによって CC へ明示して指定される場合)と，大きさの情報を含んでいる．この情報は，"main" CRL ファイルとして CLINK コマンド行に指定されたもののみ有効であり，それ以外のものでは使用されない．

コンパイル作業で外部データ領域を設定するために，－e オプションを指定するならば，5番目のセクタの第1バイトは 0xBD という値を持つ．それ以外は，その値はゼロでなければならない．第2，第3のバイトは－e オプションのオペランドとして与えられたアドレスを含んでいる．

5番目のセクタの第4，第5のバイトは，そのファイルの範囲内で宣言された外部データ領域の大きさを含んでいる（最初に，低位バイト，二番目は高位バイト）CLINK は，"main" CRL ファイルの範囲内("main" 関数を含んでいる CRL ファイル)で，これらの特別なロケーションからいつも外部データ領域のサイズを得る．"main" 関数を含んでいない CRL ファイルの中で，これらのバイトは使用されない．

---

11) 各関数名の最後の文字の7ビット目は常に1である．

2.2.3 関数モジュール

CRL ファイル内の各関数モジュールは，独立構成要素であり，関数のための再配置パラメタとコールする他の関数の名前のリストに，関数自身の2進のマシーン・コードのイメージを加えたものを含んでいる．

関数モジュールは，アドレスとは無関係で，CRL ファイルの範囲内のロケーションに対して，物理的に動き回れることを意味している（しばしばこれは，CLIB がモジュールを引きずり回すときに，使用される．関数モジュールのフォーマットは，次のようになっている．

必要関数のリスト
ボディーの長さ
ボディー
再配置パラメタ

2.2.3.1 必要関数のリスト

ユーザーが作っている関数が，他の CRL 関数をコールする場合，それらの関数リストはそのモジュールの中の最初の項目でなければならない．そのフォーマットは名前の最後の文字の7ビット目が1になっている大文字だけからなる名前の一連のリストである．ゼロバイトはそのリストを終結させる．空リスト（関数が他の関数をコールしないときのように）は，単一のゼロバイトだけである．

たとえば，関数 foobar が，Putchar，getchar，setmem と呼ばれる関数をコールすると仮定すると，Foobar の必要とする関数のリストは，次のように現われる：

```
47 45 54 43 48 41 D2 50 55 54 43 48 41 D2
g  e  t  c  h  a  r′ p  u  t  c  h  a  r′
```

53　45　54　4D　45　CD　00
s　e　t　m　e　m　(終結)

2.2.3.2　ボディーの長さ

次にくるバイトは，ボディーの長さ（バイト）である．長さを表わすワードは，低位バイト，高位バイトの順である．

2.2.3.3　ボディー

関数モジュールのボディーの部分は，0000を基点にした，関数の8080コードが含まれる．

必要関数のリストがない場合，そのコードはボディーの最初のバイトから始まる．必要とされる関数のリストが，n個ある場合，疑似ジャンプ・ベクトル・テーブル（n個のjmp命令で成り立っている）と，それを飛び超すためのジャンプ命令を，ボディーの先頭に置かなければならない．

たとえば，上記のfoobar関数のボディーの始まりは，次のようになっている．

```
jmp 000Ch
jmp 0000
jmp 0000
jmp 0000            <関数のコード>

C3 OC 00 C3 00 00 C3 00 00 C3 00 00 <関数のコード>
```

2.2.3.4　再配置パラメタ

ボディーに続いて，ボディーの範囲内の局所アドレスを参照する命令のオペランド・フィールドに対するアドレスの補正値（ボディーの開始アドレスに準ずる）である再配置パラメタが含まれる．CLINKは，このリストの中のエント

りによってさし示めされている各ワードを取り，それに対して，関数の実行時ベースのアドレスを付加する．

再配置リストの中の最初のワードは，いくつかの再配置パラメタがそのリストの中に与えられているかというカウンタである．このようにして，n個の再配置パラメタがある場合，再配置リストの長さ（長さのバイトを含む）は，2n＋2バイトになる．

たとえば，ロケーション 0x22, 0x34, 0x4f, 0x61 で配置されているような4つの局所ジャンプ命令を含む関数は，次の再配置リストを持つ．

04 00 23 00 35 00 50 00 62 00[12)]

2.3 BDS C レジスタ割当てと，関数コールの規則

2.3.1 スタック

ランタイムは，すべての局所（自動）記憶割り当てだけでなく，関数呼び出しのときにパスするすべての引数も，スタック上におかれる．

2.3.1.1 スタック・ポインタ

スタック・ポインタは，SP レジスタの中に格納され，ランタイムで，ユーザーがアクセスできる，メモリー領域のトップアドレスへ初期化される．コンパイラが，有効メモリーの終りと思う正確な場所は，リンク時に与えられるオプションによって決まる．デフォルトでは，スタック・ポインタは，CP/M BDOS のベースに初期化されるので，CCP の領域はスタックに置き換わる．したがって，プログラム実行後は，CCP をメモリーに戻すため，ワーム・ブートが行なわれる．－t オプションが使用される場合，その引数がスタックの初期値となり，－n オプションを使用すると SP は CCP のベースに初期化されるので，デ

12) 命令のアドレスは，再配置を必要としている実際のアドレスのオペランドをさし示すため，1つ増やさなければならない．

フォルトよりも少ないスタック・スペース (2K) しか得られないが，実行終了後，ワーム・ブートを**行わずに**，コマンド・レベルに戻れる．

2.3.1.2 スタックはどれくらいのスペースを取るか？

スタックには，すべての局所（自動的）データ記憶，パラメタ，リターン・アドレス，それから中間式の値を持っていて，高位メモリーから始まり，下がってゆく．

スタックの大きさは，最大の関数ネスティング時の局所データ領域の大きさに，中間式の結果などのことを考えて200～300バイト加えた値を必要とする．

ワーストケースnでの局所記憶の量を求める場合，ユーザーの使用できるフリー・メモリーの量は，次の公式によって求められる．

topofmem ( ) -endext ( )－ (n＋fudge)

ここで，fudgeの値が500ぐらいだとかなり安全である．topofmemとendextライブラリ関数は，実行中のプログラムの最高位のポインタ(スタックの位置)と外部データ領域＋１の位置のポインタを，それぞれ戻す．このようにendext ( ) の値は有効メモリーの第一バイトへのポインタである．

2.3.2 外部データ

外部データ領域は，通常，プログラム・コードの直後から始まり，外部データの終りから，スタックまでの割り当て可能領域とは，区別される．

2.3.3 関数へのエントリとリターン

Cの関数が，コントロールを受け取るとき，それはたいてい次の仕事を与えられた順で行う．：　BCレジスタをスタックにプッシュする．スタック上の局所データのためにスペースを割り当てる（必要とされる局所記憶の量によってSPを減じる)．そして，BCレジスタを一定のベース・オブ・フレームのポインタとして使用するために，新しいSP値をコピーする．SPのかわりに，BCを

使用する理由は，SPが，プッシュ，ポップなどで，激しく変動するので，変数のアドレス計算が，かなり混乱してしまうからである．

BCの古い値は，いつもコールするルーチンのためにに保持されてなければならない．

コールされた関数が，局所スタックのフレーム・スペースにnloclバイトを必要とするならば，BCをプッシュし，SPからnloclを減じ，SPをBCへコピーし，その後に，局所オフセットloffsetを持つ自動変数のアドレスは，次の公式によって簡単に計算される．

(BC)+loffset

その関数がパラメタを受け取る場合，n番目のパラメタのアドレスは，

(BC)+nlocl+2n

によって求められる．

ここでnの値は，最初のパラメタに対して，1になり，2番目のパラメタに対しては2…というようになる．この最後の公式が意味するのは，コールするプログラムは，常に逆順でパラメタをスタックにプッシュするということ，引数は，プログラムをコールする直前に，プッシュしなければならないということである．コールされた関数が，BCレジスタをプッシュした後には，現在のSPと最初のパラメタの間に，a)セーブされたBCのレジスタと，b)コールするルーチンに対するリターン・アドレスという2つの16ビット値，すなわち，4バイトのデータが含まれている．この案は，各々のパラメタが2バイトの記憶領域を取ることを必要としている．このように，バイト単位のパラメタ（Char型）はいつもパラメタとしてパスされる前に，16ビットの値（符号拡張でなく，高位バイトをゼロ化する）に変換される．

処理を完了するとき（リターン前だが），コールされた関数は，nlocl分だけSPを増やすことによって，局所記憶割当てを解除し，セーブされたBCをスタックからポップし，BCレジスタのペアをリストアしてから，コールされたプロ

グラムに戻る．

コールしたプログラムはSP値をパラメタがプッシュされる前の値にリストアする．コールされた関数は，これを行うことはできない．というのは，コールしたプログラムがいくつのパラメタをプッシュしたかを求める方法がないからである（たとえば，printf関数は，パラメタの数が一定していない．

普通，コールする関数の責任というのは，

1. 逆順で，パラメタをプッシュする．
2. 従属関数をコールし，HLかDEレジスタのいずれかに，重要な値をもっていないことを確認する(従属関数は，DEを使用する事を許され，HLの中に値をもどす)．BCレジスタは従属関数による変更からは，“安全”であると思われる．というのは，規則によって，コールされる関数は，パスされたBCレジスタ値をいつも保存しなければならないからである．BDS Cパッケージで与えられたコンパイラによってつくられたすべての関数もアセンブリ言語でコード化された関数のすべてもこれを行う必要がある．
3. 関数から戻ったら，SPをパラメタがプッシュされる前の値に戻し，HLレジスタのペアを保存するように気をつける（従属関数から戻された値を含んでいる)．スタックポインタを記憶する一番簡単な方法は，プッシュされた引数の数だけPOP命令を行えばよい．

コールされた関数の必要とされる規則は，

1. BCレジスタがコールされたプログラムにもどる前に，変更される可能性があればBCレジスタをプッシュする．
2. 局所記憶の必要があれば，必要とされる数のバイト数を，SPから減じてスタック上に適切なスペースを割り当てる．
3. 必要とあれば，BCレジスタ・ペアをベース・オブ・フレームのポインタとして使用するために，SPの新しい値をコピーする．もし，BCがセーブされなかった場合はこれを行ってはいけない．
4. 必要な処理をする．
5. 処理が終わったとき，局所のフレーム・サイズをSPに加え，局所記憶割り当てを解除する．

6. スタックから BC をポップする(ステップ1でセーブされた場合のみ).
7. HL レジスタにリターン値を格納し，コールされたプログラムに戻る.

2.4 ランタイム C.CCC 内のサブルーチン (CCC.ASM)

アセンブリ言語による関数によって使用できるランタイムパッケージの中の有用なサブルーチンがいくつかある．これらのルーチンは，局所／外部データのアクセス，パラメタのアクセスと，演算の3つに分かれている．

2.4.1 局所または外部データの取り出しルーチン

第1グループの6つのサブルーチンは，BC レジスタか，外部データ領域の先頭によって与えられるオフセットから，8ビットか，16ビットかのいずれかのオブジェクトを取り出すために使用される．ここで，オフセットは8ビットか16ビットの値として指定される．たとえば，外部データ領域（ロケーション extrns に記憶されている）から eoffset バイトのオフセットの位置に記憶されている外部変数の16ビット値を取り出すための直接的な手順は

```
lhld    extrns          ; get base of external area into HL
xi      d, eoffset      ; get offset into HL
dad     d               ; add to base-of externals pointer
mov     a, m            ; perfom 4-step
inx     h               ; indirection to
mov     h, m            ; fetch value at
mov     l, a            ;  (HL) into HL.
```

外部変数を取り出すための特別なコールを用いると，次のようになる．

```
call sdei        ;single-byte-offset, double-byte value external
db eoffset       ; indirection, with eoffset 256
```

このコードの方が，少ないメモリーで済む（正確には4バイト対11バイト）．eoffsetの値が255より大きい場合は，eoffsetをdbではなく，dwで与え，ldeiルーチンを使用する．このような，値を取り出すサブルーチンの全レパートリーに関する完全なリストや説明文については，CCC.ASMを参照せよ．

2.4.2 パラメタの取り出し

2番目のサブルーチンのグループは，スタック上に置かれる関数の引数を，AレジスタとHLレジスタに取り込む(低位バイトの値は，AとLレジスタに，高位バイトの値は，Hレジスタにそれぞれ格納される)．たとえば：ユーザーのアセンブリ言語による関数が呼ばれたばかりだとすると，サブルーチンma1tohの呼び出しは，最初の引数をHLとAに格納する．ma1toh（"Move Argument 1 toH"の意味)は，ロケーションSP+2(ユーザーの関数がSPをみるように）の16ビットの値を取り出す．ma2toh（"Move Argument 2 toH"）ルーチンへのコールは，スタックから2番目の16ビット引数を取り出し，HLとAに格納する．もしスタックからパラメタを取り出す前に，BCレジスタをプッシュするなら，そのスタック上のすべての項目はSP値からもう2バイトによってオフセットされるし，最初の引数を取り出すためには，ma2tohをコールしなければならないし，2番目の引数を取り出すには，ma3tohを，3番目はma4toh…と続く．このように，これらのサブルーチンを使用するとき，スタックの深さを考慮することは重要である．

関数の引数を取り扱うためのあまり複雑でない方法は，特にBCをプッシュしたり，スタック上で何か行なう前に，ユーザーの関数の中で，始めにarghakをコールすることである．Arghakは，RAM領域（普通，C.CCCの内で）内の14バイト・バッファに，最初の7つの関数の引数をコピーする．また，関数操作の継続している間，単にlhld操作を行なうことによって，それらの値を呼び出し可能にする．これは，arghakを使用する関数が，他のarghakを使用する関数をコールしていないことを前提とする．arghakがコールされた後は，最初の引数は絶対ロケーションarg1に記憶され，二番目はarg2…と続く．これらのシンボルは後に説明されているように，BDS.LIBの中で定義される．

2.4.3 演算サブルーチン

最後のサブルーチン・グループは，演算のグループで，これらすべてはHLとDEを引数として受け取り，HLに結果を戻す．ここではこれらの機能に関して，あまりスペースをとりたくないので，有効なサブルーチンを知るためには，ランタイムパッケージ・ソース・ファイル（CCC. ASM）を調べてもらいたい．

2.4.4 ソース・ファイル

CCC. ASMは，ランタイムパッケージのソースであり，この中に前述のルーチンのすべてが書かれている．パッケージ内のヘッダーファイルBDS. LIBはC. CCC（アセンブルされたCCC. ASM）内のすべてのエントリ・ポイントが定義されている．CSMフォーマットのソース・ファイルで，これらのエントリ・ポイントを，名前でコールできるようにするには，次の命令，

```
#include <bds.lib>
```

を含んでいなければならない．もし，ランタイムパッケージをカスタマイズするために，CCC. ASMを修正する必要があれば，BDS. LIB内の対応するアドレスも修正しなければならない．

2.5 任意ロケーションまたは，ROMで実行するコードの生成

BDS Cは，コンソール・コマンド・プロセッサ（CCP）から実行される非常駐コマンド・ファイル（ユーザー領域開始アドレス（100h）から始まる）を生成する．こういった普通の状態でいるとき，ランタイムパッケージ(C. CCC)とその専有領域は，コマンド・ファイルの最初の1500程のバイトを占め，コンパイルされたコード（“main”関数で始まる）がそのすぐ後に続く．

CP/M非常駐コマンドを生成することが目的ならば，特別な作業を必要としないが，それとは異なるロケーションで実行することのできるコード，あるいはROMの中へ置かれるコードを生成するためには，

a) ランタイムパッケージをカスタマイズすること.

b) 関数ライブラリのマシンコード化された部分を再アセンブルすること.

c) ライブラリのCコード化された部分を再コンパイルすることが必要である. そのパッケージをカスタマイズする一般的な手順は, 次のようになっている.

1. ランタイムパッケージ (CCC. ASM) を, 必要な設定を行うために, 変更して再アセンブルする. そのターゲット・コードがCP/Mのもとで操作されていない場合, 適切なEQUをゼロへセットすることによって, CP/M特有のサポート・コードを除去し, ランタイムパッケージの中に含まれているRAM領域とランタイムパッケージの両方を小さくし, ランタイムパッケージの中のCP/M用のエントリ・ポイントを未定義のままにしておく. だから, 未定義の状態でそれらのエントリ・ポイントをコールするようなコードは生成されない. また, そのパッケージの専有データ領域のためのコードの開始アドレスと, RAMのロケーションを定義するために適切なEQUを必ず設定すること.

2. CCC. ASMをアセンブルした後には, その開始アドレスがTPAのベースのところになければ, 二進のイメージを生むためにCCC. HEXファイルを単にLOADすることはできない. もし, 開始アドレスがどこかほかにあるならば, メモリーへそのファイルを読み込むために DDT か SIDを使用し, TPAのベースへそれを移動し, CP/Mをリブートし, 新しいC. CCCイメージを書きもどすためにSAVEコマンドを使用する. CCC. ASMの二進のイメージがつくられた後 (指名されたCCC. COMかあるいはそれ以外), C. CCCとなるようにそれを改名する.

3. すべてのアドレスが, 新しいCCC. ASMのアセンブリコードから得られる値とマッチするようにファイルBDS. LIBをエディットする. このステップをチェックするのに良い方法は, BDS. LIBをBDS. ASMにリネームし, それをアセンブルし, BDS. PRNとCCC. PRNの一番左側に現れる値を比較することである.

4. CASMを使用して，マシン語ライブラリファイル (DEFF2A. CSM, DEFF2B. CSM, DEFF2C. CSM) からCRLファイルを生成する．CP/M以外の環境のためにパッケージを設定する場合，アセンブル作業の前にライブラリ関数からすべてのCP/Mに関する関数をまず除去すること．ほとんどのファイルI/Oとシステム従属の関数は，DEFF2C. CSMに含まれている．
5. 非標準ロード・アドレスにコードをコンパイルするのに，CC. COMを使用するとき，新しいパッケージの開始アドレス（0x100と異なる場合）をコンパイラに知らせるために，－mオプションを使用する．必ず－mを使用してSTDLIB. Cと，STDLIB2. Cを再コンパイルする．それから，CLIBを用いてSTDLIB1. CRLと，STDLIB2. CRLのすべての関数を新しいDEFF. CRLに転送する．
6. ロード・アドレス，RAMの最高位アドレス，それぞれのターゲット，プログラムの外部データ領域のベースを－l，－t，－eのオプションでそれぞれ指定してCLINKを行なう．
7. PROMを作成する．

第 3 章

CP/MにおけるBDS C標準ライブラリ関数

BDS Cパッケージの中の，ファイルDEFF.CRLとDEFF2.CRLは，標準ライブラリ[13]のオブジェクト・コードを含んでいる．これらのCRLフォーマットのファイル内には，Cプログラムで利用できる有用なCの関数が含まれている．CLINKはコマンド行で明白に指名されているCRLファイル全部が検索された**後に**，自動的にライブラリ・ファイル[14]を検索する．このように，CLINKがライブラリの走査を行なう前に読み込まれた関数の名前と同じものがライブラリの中にみつかっても，それは使用されない．

後述するDEFF.CRLとDEFF2.CRLのすべて関数の要約では，それぞれの関数がCプログラムの中で，どのようにして**定義**されるかということを，C-ライクな表記法で解説している．この表記法では，関数が値を戻すかどうか（もし戻すならば，どんなタイプのものか)，また，関数が用いるパラメタのタイプに関する情報を提供する．大ざっぱにいうと，関数がタイプなしに使用される場合，値を戻さない(たとえば，exitとpokeは値を戻さない．多くの場合，低

---

13) DEFF.CRLは，STDLIB1.CとSTDLIB2.CからのCで書かれた関数のすべてを含んでいる．一方，DEFF2.CRLは，DEFF2A.CSM，DEFF2B.CSM，DEFF2C.CSM (CASMファシリティーを使用してアセンブルされている）のアセンブリ言語関数すべてを含んでいる．

14) 必要とあれば，そこにすべてのライブラリ・ファイルをもっていなくても，どんなドライブやユーザー領域の中からでも，リンクは行われ，ユーザーは任意のCP/Mディスク・ドライブとユーザー領域の中でライブラリ・ファイルを検索するためにCLINKを設定してもよい．

位8ビットのパラメタだけが使用され，char 型の値が関数へパスされるか，明白な宣言の欠けているパラメタは，常にタイプ int である．unsigned タイプのメモリー・ポインタか，文字ポインタかを区別するのは，ちょっとめんどうである．16ビットのメモリー・アドレス・パラメタの位置で char 変数をパスしようとしない限りは，パラメタの型宣言がコールしているプログラムの中にあるかどうかにかかわらず，事はうまく運ぶであろう．

ライブラリ関数がCプログラムの中で使用される前に，それを実際に宣言する必要のある場合がいくつかある．int 以外の値を返す関数が式の内部で使われているときがそうであり，その場合，その式を妥当な int 型以外のものにする必要がある(たとえば，ポインタ演算におけるように)．ある関数を宣言することが妥当であるとき，あるいは不必要なときをはっきりさせるのに，ちょっとした経験が役に立つだろう：　こういった決断の多くは，スタイルとポータビリティ（移植性），あるいはそのいずれかの問題である．

DEFF.CRL と DEFF2.CRL で有効な主な関数全部の要約について述べる．

3.1　汎用関数

char csw( )

は，コンソールスイッチ・レジスタ(いくつかのメインフレームでポート 0xFF)のバイト値（0 －255）を戻す．

exit( )

は，オープンされているファイルをクローズし，実行プログラムを終了し，CP/M をリブートする．バッファ付き出力のために，オープンされたファイルに対しての fflush は自動的に行われない．

int bdos (c, de)

は，まず値cをCレジスタに値deをレジスタ・ペアーDEにセットして，標準BDOSシステムのエントリ・ポイント（ほとんどのシステムでは，ロケーション0005h）をコールする．BDOSから返ってくる値は，16ビット値でHLレジスタに格納される．CP/Mシステムでは，レジスタにBDOSによってもどされたAレジスタの値が，HレジスタにはBDOSによって戻されたBレジスタの値（8ビットのリターン値ではゼロ）がそれぞれ格納される．CP/Mでないシステム（たとえば，SDOSなど）との不適合な点についての詳しいことは，付録の"多方面にわたる注意点"を参照のこと．

char bios (n, c)

は，まずはじめに値cをBCレジスタにセットして，BIOSのジャンプベクトル・テーブルの中のn番目のエントリをコールする．ここで，nというのは最初のエントリ(BOOT)は0であり，2番目の(WBOOT)は1，3番目(CONST)は2，……となる．リターン値は，BIOSコール終了後のAレジスタの値となる．コールド・ブート機能（nが0）は実際には使用すべきではない．というのは，CCPはスタックによって消されているので，たぶん，システムをこわしてしまうだろうからである．DEの中のパラメタを必要とし，HLの中にその結果を戻すBIOSコールがいくつかある．これらのコールのためには，biosh関数(次に説明する）を使用する．

unsigned biosh (n, bc, de)

は，まずはじめに値bcをBCレジスタへ，また値de
をDEレジスタへセットし，BIOSジャンプ・ベクトル・テーブルの中のn番目のエントリをコールする．結果は，BIOSコールによってHLレジスタの中に戻される値になる．

char peek (n)

は，メモリー・ロケーションnの内容を戻す．多くの連続的ロケーションを調べる必要のあるような応用においては，peekを使用するよりも文字ポインタで

間接的手段を用いた方がより効率的である．この関数は，ポインタを宣言したり，アドレスをそれに割り当てたり，単にアクセスするのに間接的手段，たとえば，単一メモリ・ロケーションを使用したりするのにはやっかいになるときなどの臨時の場合に使用する方がよい．

poke (n, b)

は，メモリー・ロケーションnに，bの低位8ビットを書き込む．これは，

```
* n=b;
```

のように，ポインタを使用してより効果的に達成されうる．(ここで，nは文字ポインタである.)

inp (n)

ポートnから8ビットの値を取り出して戻す．メモリーマップされた入力機器のためにはpeek関数を使用する．

outp (n, b)

は，8ビット値bをポートnへ出力する．メモリーマップされた出力機器のためにはpoke関数を使用する．

pause ( )

は，CP/Mのコンソール入力があるまでループを回りつづける．しかし，入力された文字は，取り出されない．pauseを再び使用する前にgetcharによってキー入力のステータスをクリアする必要がある．リターン値はない．

sleep (n)

は，4MHzでn/20秒の間，2MHzでn/10秒の間，実行を停止する．ループが完結する前に打ち切る方法は，コントロール-Cをタイプすることである．これはプログラムを打ち切り，コマンド・レベルへ戻る．リターン値はない．

int call (addr, a, h, b, a)

は，次のようにCPUレジスタをセットし，ロケーション addr で示されるマシーン・コードのサブルーチンをコールする．

HL < - h；
A < - a；
BC < - b；
DE < - d；

リターン値はサブルーチンがレジスタ HL に格納した値となる．サブルーチンはもちろん，スタック規則を維持すること．

char calla (addr, a, h, b, d)

は，サブルーチンによって戻された結果が(HL のかわりに) A レジスタの値になるということを除けば，call 関数と同じである．

int abs (n)

は，n の絶対値を戻す．

int max (n1, n2)

は，2つの整数値の大きい方を戻す．

srand (n)

n が 0 以外なら，この関数は，内部シード値を n へセットすることによって疑似乱数生成器を初期化する．n がゼロなら，srand がユーザーがキャリッジ・リターンをタイプするようにメッセージをプリントし，その間，内部で，シード値をカウント・アップし続ける．キーがユーザーによって打たれるとき，そのカウントの現在値はランダムなシードを初期化するために，使用される．ユーザーのタイプした文字は失われ，ステータスはクリアされる．

```
srand1 (string)
char * string;
```

は，前もって用意された"Hit return after a few seconds"というメッセージのかわりに，与えられたストリングがプロンプトとして使用されるということを除けば，Srand (0)と同様である．しかし，Srandとちがって，そのプロンプトに答えて，ユーザーがタイプした文字は失われないので，getcharをコールしてその文字をサンプルし，コンソールステータスをクリアする必要がある．

```
int rand ( )
```

は，SrandかSrand1によって初期化された疑似乱数順で，次の値（範囲：0 <rand ( ) <32768）を戻す．0からn-1の間の値を得るためには，次式

```
rand ( )   %   n
```

を使用すればよい．

```
nrand(−1, S1, S2, S3)
nrand(0, prompt-string)
int nrand (1)
```

これは，コーラント数理科学研究所で利用される．CDC6600乱数生成器と競合するポール・ガンズ教授によって書かれた新しい，そして"より良い質"の乱数生成器である．初期化のメカニズムは，srandとsrand1の規制との適合性のためにつけ加えられた．最初のフォームとは，S1，S2，S3(intかunsigned)によって指定された48ビットのデータと同じように内部の48ビット・シードをセットする．二番目のフォームは，srand1関数と全く同じように作動する：Prompt-stringによって示された文字列をコンソールにプリントし，それから，マシーンはコンスタントに内部の16ビット，カウンターを増加させながら，ユーザーからのキー入力を待つ．文字がタイプされるとすぐに，そのカウンター

の値は48ビットのシードへと変換される．コンソール入力はクリアされないので，その後にgetcharコールしてタイプされた文字を実際に取り出し，コンソールステータスをクリアすることを行なう．3つめのフォームは

0 ＜ nrand (1) ＜ 32768

の範囲で，ランダムな値を戻す．

nrandによって維持される内部シードは，srand, srand1, randによって使用されるシードとは区別される．これは，ランタイムパッケージ・データ領域の範囲内のrseedとラベルされた領域の最初の32ビットを使用する．Nrandはそれ自身の明白な内部シードを維持する．

```
setmen (addr, count, byte)
char byte, * addr ;
```

は, addrで始まるメモリーの一連のバイト数countを値byteへセットする．これは，配列とバッファ領域を早く初期化するのに効率的である．

```
movemen (source, dest, count)
char * source, * dest ;
```

は, sourceアドレスからcountバイトのブロックをdestアドレスへ動かす．この関数は，ブロック・ムーブを先頭から先頭までするか，末尾から末尾へするかを自動的に検知し, ソースと行き先の領域の構成を正しく操作する. Z80プロセッサで実行している場合，Z80の“ブロック・ムーブ”命令が使用される．8080か8085で実行している場合，普通の8080の命令が使用される．こういったことは，すべて自動的に行われる．

```
qsort (base, nel, width, compar)
char * base ;
```

```
int (* compar) ( );
```

は, baseから始まるwidthバイトのnel個のデータを"シェル・ソート"する. comparは, 2つのアーギュメント(例えば, x, y)を比較する関数のポインタであり, リターン値は次のようになる.

もし ＊× ＞ ＊y ならば 1
もし ＊× ＜ ＊y 〃 -1
もし ＊× == ＊y 〃 φ

エレメントは, 昇順でソートされる.

```
int exec (prog)
char * prog ;
```

は, プログラムProg.COMをチェイン（ロードし, 実行）する. progは, チェインされるためにファイルを指定する0でくぎられたストリング・ポインタでなければならない("COM"はネームで表わされる必要はない). ストリング定数("foo"など)は, ポインタとして扱われるので, 使用可能である. execされるプログラムが, Cコンパイラによって生成され, それが外部変数をexecしたプログラムと共有することが必要な場合, それは同じアドレスに共通外部データを配置するために, CLINKオプション－eを用いてリンクされているべきである. －eオプションの正しい使い方については, CLINKの説明書をみよ. execコールによって, オープン・ファイルの所有権が移ったりしない. この案のもとで唯一共有できる資源は, 上記の外部データである.

エラーのとき－1を戻すが, 戻ってくるというのは, すなわエラーなのである.

```
int execl (prog, arg1, arg2, ……, 0)
```

```
char * prog, * arg1, * arg2, ……
```

は，argcとargvのメカニズムを用いて，1つのC.COMファイルから別のものへとチェインする．progは，チェインされるCOMファイルの名前を示す0で区切られたストリングでなければならない（".COM"は名前の中に現われる必要はない）．各引数もまた0で区切られたストリングでなければならず，最後の引数はゼロでなければならない．Execlは，与えられたパラメタから，コマンド行をつくることによって作動する．また，ユーザーがCP/Mのコマンド・プロセッサに，そのコマンド行をタイプしたように作動する．たとえば，

```
execl(" foo ", " bar ", " zot ", 0);
```

は，CP/Mのコマンド行

```
A〉 foo bar zot 〈cr〉
```

が接直タイプされるのと同じ意味を持つ．あいにく，CP/M内部コマンド（" dir ", " era "などのように）はexeclで呼び出されることはできない．与えられた文字例引数から，掲載されたコマンド行全体の長さは，ほぼ80文字を越えてはならない．構成されたコマンド行が，この長さを越えた場合，その影響のメッセージはコンソールにプリントされ，プログラムは打ち切られる．エラーのとき，－1が戻される（戻ってくる場合は，なんらかのエラーがあると思われる）．

```
execv (filename, argvector)
char * filename ;
char * argvector [ ] ;
```

この関数は，引数の変数を含めてチェインするようにする．パラメタ・テキストが，個別に指定されるかわりに，配列の中で指定されることを除けば，execlと同様である．argvectorパラメタは，ストリング・ポインタの配列に対するポ

インタでなければならない．ここで，ストリング・ポインタは次の引数に対してさし示し，また，最後のポインタのポインタ値は，0でなければならない（0で区切られたストリングポインタではない）．リターンがあれば，それはエラーを意味しているのだが，エラーのときは，－1を戻す．

```
int swapin (filename,addr)
char * filename;
```

は，filenameで示される0で区切られたストリングで示されるファイルを，メモリーロケーションaddrからロードする．ファイルの大きさが，メモリーサイズより大きいかどうかはチェックされないので，どこでユーザーがそれをロードするのかに注意すること！　たとえば，この関数は後の実行のために間接テキスト手段によるオーバーレイ・セグメントを関数へのポインタからロードするために使用される．ファイルを読み取っていると，エラーがあれば－1を戻す．コントロールはロードされたファイルへ移らない．

```
char * codend( )
```

は，ルート・セグメントのプログラム・コードの末尾につづく最初のバイトのポインタを戻す．CLINKの－eオプションが，外部データを配置するために使用されなければ，これはふつう，外部データ領域の始まりになるであろう（下記のexterns関数をみよ）．

```
char * externs( )
```

は，外部データ領域の始まりのポインタを戻す．－eオプションがCCとCLINKあるいはその一方で使用されたものでなければ，この値は，codend関数によって戻されたものと同じものになる．

```
char * endext( )
```

は，外部データ領域の末尾につづく最初のバイト・ポインタを戻す．これはそ

の領域のスタートになり，そこからsbrk関数は空きメモリーを得る．

```
char * topofmen ( )
```

は，ユーザー・メモリーの最後のバイトのポインタを戻す．これは普通，スタックのトップになり，BDOS (－nオプションがリンク時に与えられない場合) のすぐ上か，CCP (－nがリンク時に使用される場合) のすぐ上のいずれかにある．topofmemによって戻された値は，リンク時の－tオプションの使用には影響されない．

```
char * alloc (n)
```

は，ポインタを長さnバイトのメモリーのフリー・ブロックを戻す．あるいは，メモリーのnバイトが有効でない場合，0を戻す．これは大体のところカーニンハン & リッチーの本の第8章からきている記憶割り付け機能であり，タイプ位置合わせの制限がないために単純化されている．詳しいことはその本を参照のこと．標準ヘッダーファイルBDSCIO. Hは，allocとfreeのペアを使用しているプログラムのすべてのファイルの中で，#includeされていなければならない．というのは，その中で宣言された重要な外部データがいくつかあるからである．

```
free (allocptr)
char * allocptr ;
```

は，alloc関数によって割りつけられたメモリーブロックを解放する．ここでallocptrはallocに対する先の呼び出しによって得られた値である．Freeは先のalloc呼び出しの逆の順番でコールされる必要はない．というのは，リンクデータの構造は，割り付け除去のどんな順番も許容できるからである．allocの呼び出しによって，先に得られた値以外の引数を用いてfreeを呼び出してはいけない．

char * sbrk (n)

これは，低レベルの記憶割り付け関数であり，メモリーを得るために alloc によって使用される．これはnバイトのメモリーのポインタを戻す．あるいはnバイトが有効でなければ－1を戻す．最初の sbrk コールは，外部データの末尾のすぐ後につづくメモリーロケーションのポインタを戻す．sbrk は，割り付けられたロケーションが現在のスタック・ポインタの値に近くなるまで，その後の各々のコールが隣接するブロックを戻し続ける．"危険"な接近はデフォルトでは1000バイトと定義される．この値を変更するためには次の関数をみよ．プログラムの中で alloc と free の関数を使用することを計画しているか，また，スクラッチ・スペースに有効な割り付けからいくつかのメモリーを免除したい場合，alloc のかわりに必要なメモリーを要求するために，sbrk ( ) を使用しなさい．sbrk コールはいつでも使用できる（すでになされた alloc と free コールとは無関係である）．

rsvstk (n)

この関数によって，記憶割り当て機能は，割り当てられた領域の末尾からスタック・ポインタの現在の値までの間のnバイト以上の割り当てコールを拒否するようにする（そのスタックは，ハイ・メモリーより低くなることを忘れずに）．Rsvstk は,sbrk か alloc の呼び出しを行う前にコールされるべきである．rsvstk が使用されない場合，記憶割り当ては1000バイト以上，スタックに近づかないように自動的に設定されている（"rsvstk (1000)"コールがなされたかのように）．

```
int setjmp (buffer)
char  buffer [JBUFSIZE];
longjmp (buffer, val)
char buffer [JBUFSIZE];
```

setjmpがコールされるとき，現在のプロセッサの状態は，与えられたバッファの中にセーブされ(記号変数JBUFSIZEはBDSCIO.Hの中で定義される)，0の値が戻される．あとに続くlongjmpコールが，現在の関数かあるいは，より低いレベルの関数のいずれかどこかにおいてなされるとき，CPUの状態は元のsetjmpコールがなされたときにもっていたものに置き換わる．プログラムは元のsetjmpコールへ“リターンする”ことによって実行を再開し，そのとき，値val(longjmpにパスされたように)が戻される．プログラムがsetjmp初期化コールと処理の以降とを区別できるように，longjmpへパスするvalの値はゼロ以外であるべきである．setjmp/longjmpの典型的な使い方としては，数レベルのネスティングされた関数から順番に各レベルを通してリターンする必要もなく，ネスティングを退出することである．たとえば，特殊な出口ルーチン（いうならば，DIO.Cパッケージのdioflush）がいつも行なわれることを確実にすることである．

3.2 文字の入出力

3.2.1 コンソールI/Oを直接行うCIO関数パッケージ

標準ライブラリで与えられたgetcharとputcharの関数（後で説明されている）は，コンソールの絶対的なコントロールを認めない．かわりにそれらは，初期化や特殊な考えを要求せずに規則どおりの応用にとって最も有効であるように設計されている．システムコンソールの装置から受け取られる，また，そこに送られるすべての文字に対して，完全なコントロールをできることが重要であるようなアプリケーションをつくる場合，BDS C V1.50パッケージ内のCIO関数パッケージ(CIO.CSM)をアセンブルし，使用しなさい．CIOは，コンソール・インタフェースの特性を動的に抑制するための新しいttymodeだけでなく，getchar，putchar，kbhitといった関数の代替版も含んでいる．

int getchar ( )

は，標準入力ストリーム（CP/M コンソール入力）からの次の文字を戻す．コントロールCがタイプされると，CP/M をリブートする．キャリッジ・リターンは CR-LF をコンソールへ出力し，ニューライン（'＼n'）記号を戻す．コントロールZが入力されると値－1が戻される．リターン値－1を整数として認識するには，getchar からのリターン値を整数として扱わなければならない．（文字ではなく）かわりに，ユーザーが getchar を char 型の値を戻すように宣言する場合，あるいは，そのリターン値を文字変数へ割り当てる場合，値255はコントロールZを検出するためにチェックされるべきである．しかし，この場合，実際のデータ値255と，EOF マーカーとを識別できないことに注意すること．

char ungetch (c)

は，文字 c が次の getchar の呼び出しによって戻されるようにする．連続的な getchar 呼び出しは，一つの文字だけしか"返せない"であろう．普通，ゼロが戻される．最後の getchar 呼び出しのとき，ungetch された文字がすでにあったとしたら，その文字の値が戻される．

int kbhit ( )

標準入力があれば（キーボードが押された），真（ゼロ以外）を戻し，その他の場合は偽（ゼロ）を戻す．実際に入力が取り込まれることはない．それを行うには，getchar の呼び出しを必要とすることになる．ungetch 関数が最後の getchar の呼び出し以来，コンソールへ文字を押しもどすために使用された場合，kbhit もまた，真を戻すことに注意せよ．

putchar (c)
char c;

は，文字 c を標準出力（CP/M コンソール出力）へ書き込む．ニューライン（'＼n'）記号は，出力のとき CR-LF の組み合わせへと拡張される．コントロ

ール-Cがputcharコールの間にコンソールから入力された場合,プログラムの実行は中止し，コントロールはコマンド・レベルへ戻る．そして，ユーザーがコントロールCをタイプすることによって，コンソール出力(putcharコールによって）を行うプログラムを打ち切られるようにする．与えられたputchar関数はコンソールに文字を出力し，コンソールからの入力をチェックするためにBDOSを使用するので，特別なCP/Mフロー・コントロール記号（コントロールS）が認識され，putcharによってなされたプリント・アウトを一時停止するのに使用できる．

```
putch(c)
char c ;
```

出力の間，コントロールCの入力をチェックしないことを除けば，putcharと同様で，割り込みを有するシステムでは，コンソール出力の間，タイプ・アヘッドを認める．もし，ユーザーがこの特徴を好み，putchコールへマップされたすべてのputcharコールを求めているなら，BDSCIO.Hのヘッダーファイルのどこかに，次のプリプロセッサ命令を置けばよい．

```
#define putchar putch
```

そして，すべてのプログラムの中には，必ずヘッダーファイルを含んでいるようにしなさい．

```
puts (str)
char * str;
```

は，0で終結された文字列strを標準出力へ書き出す．ニューラインは自動的に付加されない．

```
int getline (strbuf, maxllen)
```

```
char * strbuf;
```

は, maxlenの文字数だけコンソールからテキストの入力を行う. リターン値は入力された行の文字数である. リターンのとき, 入力行はNULL(0)によってのみ終結される. だから空の行は, 長さ0をもっていることになる (ユーザーがキャリッジ・リターン記号のみをタイプしたとき). バッファの中には, ニューライン記号は戻されない. これは"カーニンハンとリッチー"で述べられたgetline関数からの偏差である.

入力された文字の数が, 与えられた最大数マイナス1に達する (終結する空白の余裕をとるため) 場合, その行は完全と思われ, コントロールはキャリッジ・リターンがタイプされるのを待たずに, 呼ばれたプログラムへすぐに戻る. これはBDOSファンクション10がコンソール入力を読むのに使用されるから起こるのである.

```
char * gets (str)
char * str ;
```

は, コンソールからの入力を, ロケーションstrのメモリーからストアしNULLで終結しておく. 入力行を終結するために, ユーザーによってタイプされたニューラインはバッファへコピーされない. ニューラインの前の文字は, 終結するNULLの後にすぐ続けられる. リターン値は, strの始まりへのポインタである. 与えられたバッファのサイズは, 終結するNULLのためにユーザーが入れられると思った最長のストリングよりも, 少なくとも1バイト長くなければならない.

注意として, バッファを大きくすることを推奨する. というのは, ここでオーバーフローすると, たぶんそのまわりのデータもほとんど破壊するかもしれないからである. 入れられた文字の数が135に達する場合, その行は終結したと思われる.

```
printf (format, argl ,arg2, …)
char * format;
```

書式付きプリント関数で，出力は標準出力である．標準バージョンでサポートされる変換記号は：

d	十進整数フォーマット
u	符号なしの整数フォーマット
c	単一文字
s	ストリング（NULLで終結）
o	8進フォーマット
x	10進フォーマット

各変換は，

% [−] [[0]w] [.n]　変換文字〉

というフォームでできている．ここでwはフィールドの幅を指定し，n（あるならば）はストリング変換時の文字の最大数を指定する．wのデフォルトの値は1である．左づめをさせるハイフンがパーセントサインの後につづけて指定されなければ，そのフィールドは右づめで出力される．wの値がゼロによって先行される場合，スペースのかわりに0がフィールドのパッドに使用される．16進のアドレスをプリントするときなどに有用である．Bob Mathiasの浮動小数点パッケージで使用される浮動小数点値のためのeとfの書式変換を組み込んだprintfは，ファイルFLOAT.Cに含まれる．

```
int scanf (format, aegl, arg2, …)
 char * format;
```

書式付き入力，これはprintfに類似しているが，反対の操作をする．%u変

換は認識されないので，符号つきの，または符号なしの数字の入力の両方に％dを使用すること．代入禁止文字（＊）は動作するが，フィールド幅仕様はサポートされない．scanf に対する引数は絶対にポインタでなければならない！！！！！

入力文字例（書式付き文字列での％s 変換仕様によって示される）は，書式付き文字列の中の％s に続く文字が走査されるときだけ終結されることに注意すること．リターン値は，うまく割り当てられた項目の数である．scanf と printf についての詳しいことは，カーニンハン & リッチーの本の145～150ページ（和文では158ページ）を参照せよ．

3.3 文字列と文字の処理

```
int isalpha (c)
char c;
```

は，文字 c がアルファベットならば真（ゼロ以外），さもなければ偽（ゼロ）を戻す．

```
int isupper (c)
char c;
```

文字 c が大文字ならば真，さもなくば偽を戻す．

```
int islower (c)
char c;
```

記号 c が小文字ならば真，さもなくば偽を戻す．

```
int isdigit (c)
```

char c;

記号cが十進数ならば真，さもなくば偽を戻す．

int toupper (c)
char c;

cが小文字なら，cを大文字にしたものが戻される．さもなくばcが戻される．

int tolower (c)
char c;

cが大文字なら，cを小文字にしたものが戻される．さもなくばcが戻される．

int isspace (c)
char c;

文字cが"ホワイト・スペース"（ブランク，タブかニューライン）なら真を戻す．さもなくば偽を戻す．

sprintf (string, format, arg1, arg2, …)
char * string, * format;

出力がコンソールのかわりに，stringによってさし示されたメモリー・ロケーションへ書き込まれることを除けば，printfと同様である．

int sscanf (string, format, arg1, arg2, …)

```
char * string,* format;
```

テキストが，コンソールのキーボードのかわりに string によってさし示された文字列から走査されることを除けば，scanf と同様である．うまく割り当てられた項目の数を戻す引数は，割り当てを必要とする対象に対するポインタでなければならないということを忘れずに．

```
strcat (s1, s2)
char * s1,* s2;
```

は，NULL で終結する文字列 s1 の末尾へ s2 を連絡する．もちろん，s1 は，連絡できる十分な余裕がなければならない．

```
int strcmp (s1, s2)
char * s1,* s2;
```

(s1> s2) の場合，正の値を，(s1=s2) の場合ゼロを，(s1 <s2) の場合負の値を戻す．標準 ASCII コードシーケンスが比較のために使用されるので，文字列がアルファベット順でくるなら，それは“より大きく”なる．

```
strcpy (s1, s2)
char * s1,* s2;
```

文字列 s2 をロケーション s1 へコピーする．たとえば，foo と名付けられた文字配列を，文字列 barzot へ初期化するには，

```
strcpy (foo, " barzot ");
```

となる．正しい添え書なしで，配列名が値として使用されるべきでないので命令文

```
foo= " barzot ";
```

は正しくないということに注意せよ．また，式" barzot "は文字列そのものではなく" barzot "に対するポインタを，その値としてもっている．だから後の構造が作動するために，fooは配列としてでなく文字に対するポインタとして宣言されなければならない．だが，このアプローチは危険である．というのは，fooの末尾に何かを付加するのに自然な方法は，

```
strcat (foo, " mumble ");
```

では，" barzot "につづく6バイトをオーバーライトしている（" barzot "が関数のコード内で記憶されるときはいつでも）ので，たぶん，ひどい結果になるだろう．可能な解決法が2つある．ユーザーはfooで割り当てられうる文字の一番大きな数を計算し，初期の割り当てを適切な数のブランクで埋め込む．たとえば，

```
foo= " barzot      ")
foo [6]=NULL;
```

あるいは，

```
char work [200] ,* foo;
```

を用いて十分なサイズでの文字配列を宣言することができる．そのときは，

```
foo=work;
```

と宣言することによって配列のポインターをfooにもたせる．また，

```
strcpy (foo, " numble-fraz ");
```

を使用してfooを割り当てる．

```
int strlen (string)
char * string;
```

string の長さ（NULL 記号を検出する前までの文字数）を戻す．

```
int atoi (string)
char * string;
```

ASCII ストリングをその一致する整数（あるいは符号なしの）値へ変換する．受け入れられるフォーマットは，いくつものホワイトスペースを（スペース，タブ・ニューライン）含んでいてもよい．そして，任意のマイナス・サインと，十進数の連続する文字列によって続けられる．最初の数字以外のものは走査を終結する．正当な値がみつけられなければゼロの値が戻される．

```
initw (array, string)
int * array;
char * string;
```

これは，整数配列の初期化を行うための関数である．Array は初期化されるために配列へさし示すべきである．また，string はコンマによって区切られた整数値の ASCII ストリングへさし示すべきである．たとえば，UNIX C の

```
int values [5] = {-23, 0, 1, 34, 99 } ;
```

は，次の文に取って替る．

```
int values [5] ;
```

また，どこか適切な場所に，命令文

```
initw (values, " −23, 0, 1, 34, 99 " ) ;
```

を挿入する．

```
initb (array, string)
char * array,* string;
```

は，文字配列のための前記のinitw関数の等価である．stringはinitwと同じフォーマットのものである．しかし，各種の低位8ビットがarrayの連続バイトへ割り当てるために使用される．この関数は，文字ポインターの配列を初期化するために使用されないことに注意しなさい．これは本当に，“文字”向きにできているのではないが，“文字”変数の列の中で値をもっている10進の整数向きにできていて，このようにして文字として記憶される．

注意： UNIX Cプログラムは，時々負の値を文字変数へ割り当てる．というのは，UNIX C文字変数は符号つきの8ビットの量である．BDS Cでは，記号変数は常に符号なしの値をもっていて，負の値は16ビットのint変数へ意味深長に割り当てられるだけである．

```
int getval (strptr)
char ** strptr;
```

initwとinitbからの副産物であり，コンマによって区切られたascii値の文字列に対するポインターへ，ポインターを与えられている．getvalは文字列の中でさし示された現在の値を戻し，次の値へとさし示すためにポインターを更新する（なぜ，strptrは記号に対する単一ポインターであることができないのか？[15]).

---

15) なぜならば，文字列内の文字ポインターは，getvalルーチンによって変更されるからである．関数によって変更されるような値は，その値に対するポインターを通して操作されなければならない．このように，“文字ポインター”は文字ポインターへのポインターを通して操作されなければならない．

終結バイト(0)を検出すると，−32760の値が戻される．Initwはこのように−32760の値を受け入れない．その値を使用する必要があるならSTDLIB.C内にある，終結する値を他の値へと変更しなさい（ユーザーはgetvalとinitwも変更しなければならないだろう）．

3.4 ファイルI/O

3.4.1 BDS Cのファイル入出力の関数について

BDS CライブラリのファイルI/O関数は大きく分けて2つに分類される．原始（低レベルの）関数は1つのセクター・サイズを単位として，ディスクへデータを書いたり，ディスクからデータを読んだりする．バッファ付きI/Oは1度に1バイトとか，1度テキストの1行といったようにユーザーが，より使いやすい大きさでデータを処理できるものである．低レベル関数が最初に述べられ，次にバッファ付き関数を述べる．

3.4.2 ファイルネーム

関数が引数としてファイルネームを用いるときは，そのファイルネームはその値が文字式か，あるいは文字列のいずれかのポインターでなければならない．正しいファイルネームは大文字や小文字を含んでもよいが，文字列の中にホワイト・スペースがあってはいけない．

3.4.2.1 ドライブ名

ファイルネームは，特別なCP/Mドライブを指定するために，"d："の形式の任意のディスク名を指定できる．デフォルトは現在ログされたディスクである．文字dは，AからZまでの1文字のドライブ名である（ユーザーのシステムの既存の論理的装置に依存する）．

3.4.2.2 ユーザー領域

"#/"で表わされる任意のユーザー領域の指定もまたファイルネームの先頭に指定することができる．ここで，#は0から31までの範囲の十進数である．省略されている場合，現在のユーザー領域が指定されたとみなされる．ドライブ名とユーザー領域の指定の両方が与えられる場合，ユーザー領域の指定がまず最初にならなければならない．たとえば，ドライブCの領域7で"foobar. zot"と指定されたファイルをオープンするためには，

```
open ( " 7/c:foobar. zot " , mode);
```

とする．ある奇妙な文字（コントロールコードのように）がファイルネームの中で検知される場合は，ファイルネームは受け付けられなくて，エラー値が関数によって戻される．これは，その名前が印字抑制している文字をもつようなファイルをオープンすることによって起こる問題をいく分軽減する．が，複雑な式などで，ファイルネームを構成するときは気をつけること．

3.4.3 エラーの処理

3.4.3.1 Errno/Errmsg関数

BDS CのV1.50には，新しいファイルI/Oのエラーの診断機能が組み込まれている．エラーが起こったときはいつでも，－1（ERROR）値が問題の関数によって戻される．これが起こるときはいつでも，エラーについてのより詳しい情報を与える特別エラー・コード・ナンバーを戻すためにerrno関数を呼び出せる．errnoによって戻された値をerrmsg関数へ渡す場合，errmsgはどんなエラーが起こったのかを正確に説明する文字列のポインターを戻す．次にあるのは，この機能の使い方の例である．これは，write命令の間に起こるエラーを診断するものである．

```
if (write (fd, buffer, nsects )!=nsects) {
```

```
        printf ( " Write error:% s\n " , errmsg (errno( ));
        …  /*この他のリカバリー処理*/
    }
```

write関数は，－1（ERROR）が単なるエラーではない．writeは書き込まれたセクターの数を戻す．これはリターン値が書き込むように指示されたセクターの数に等しくない場合，エラーとみなされる．

3.4.3.2 ランダム・レコードのオーバーフロー

oflow関数は，大きなファイルの読み取り／書き込みの際にオーバーフローが発生したかを検知するためのものである．オーバーフローは，ユーザーがファイル内の65535番目のセクターを過ぎて読み／書きしようとする場合に発生する．

3.4.4 原始ファイルI/O関数

```
int open (filename, mode)
    char * filename;
```

もし，modeがゼロならば，ファイルは入力のために，modeが1ならば出力のために，そしてmodeが2ならば，入力と出力のためにオープンされる．リターン値はファイル記述子であり，－1ならばエラーである．ファイル記述子は，read，write，seek，tell，abortそしてcloseで使用する．

```
int creat (filename)
char * filename;
```

は，まず最初にその名前を持つ既存のファイルを削除し，与えられた名前を持つ空のファイルをつくり，新しいファイルは自動的に読みと書き込みの両方のためにオープンされる．リターン値であるファイル記述子の値はread，write，

seek，tell，fabort，close といった関数を使用するときに必要となる．値－1 が戻ってきたらエラーである．

```
int close (fd)
```

は，ファイル記述子 fd によって指定されたファイルをクローズし，別のファイルの使用のために fd を解放する．ディスク・アクセスは，書き込みのためにオープンされたファイルがクローズされるときに起こるだけである．そのファイルが読み取りのためにだけオープンされた場合，fd が解放されるが，ファイルをクローズするための CP/M コールは行なわれない．close はバッファ付き I/O ファイルのためには，使用されるべきではなく，かわりに，fclose を使用すること．エラーのときは－1を戻す．すべてのオープンされたファイルは main 関数から，ランタイム・パッケージへ戻る際に，あるいは exit 関数が呼び出されたときに，自動的にクローズされる．オープンされたファイルがクローズされないようにするためには，fabort 関数を使用すること．

```
int read (fd, buf, nbl)
char * buf;
```

は，ファイル記述子 fd で指定されるファイルから，buf で指定されるメモリーの中へ nbl ブロック（各々128バイトの長さで）を読み込む．そのファイルと関連している r/w ポインターは，読み込まれたデータの直後に配置される．read の関数の呼び出しで読み込まれるデータは，最後の read あるいは write 関数が読み／書き終えた場所から始まる．seek 関数は，r/w ポインターを変更するために使用される．リターン値は実際に読み込まれたブロックの数になり，EOF のときは0を，エラーのときには－1を戻す．ファイルの中で実際に残されたのが x ブロックだけのときに，n ブロックのデータを求める場合（ここで $0<x<n$），read は x をそのリターン値とし，次の呼び出しのときには0を戻し（seek は使用せず），また，次に起こる呼び出しのときは－1を戻す．

int write (fd, buf, nbl)
chaer * buf;

bufで始まるメモリーからnblブロックを，ファイルfdに書く．writeの関数の呼び出しで書き込まれるデータは，seekがr/wポインターを変更するために使用されなければ，最後のreadかwriteの関数が読み／書き終えた地点からひき続いて始まる．致命的なエラーのときは－1を戻す．あるいはリターン値がエラー(－1)でなくても，うまく書かれたレコードの数がnblと異なる場合，それはたぶんディスクスペースを使い果たしたということを意味する．これはエラーとしてみなされるべきである．

int seek (fd, offset, code)

は，ファイルfdに関連するr/wのレコード(セクター)ポインターを変更する．seekはcodeがゼロの場合，r/wポインターをレコードoffsetへセットする．codeが1の場合，r/wポインターを現在の値プラスoffsetへセットする．(offsetは負でもよい)．codeが2の場合は，r/wポインターをファイルの終わりレコードのナンバー，プラスoffsetへセットする．このタイプのシークでは，offsetの値がファイルの中の既存のレコードへさし示されるために負でなければならない．codeが2で，offsetが0の場合，r/wポインターはファイルに対してアペンドするために，準備される．

リターン値－1はEOF (codeが2と同じ) からのシークの間に，ある種のBDOSのエラーが発生したことを示す．errno関数は起こったエラーの種類に関して，より詳しい内容を与える．seekはバッファ付きI/Oのためにオープンされたファイルに対して行なってはいけない．

int tell (fd)

は，fdに関連するファイルのr/wポインターの値を戻す．この数は0から始まり，ファイルへ書き込まれる，あるいはファイルから読み取られる次のセクターを示す．

```
int nulink (filename)
char * filename;
```

は，ファイルシステムから指定されたファイルを削除する．注意深く使用すること！

```
int rename (old, new)
char * old, * new;
```

は，ファイルのリネーム（改名）を行う．renameで指定されるファイルは，オープンされていてはいけない．エラーのときは－1を戻す．

```
int fabort (fd)
```

は，ファイル記述子 fd をクローズせずに解放する．ファイルが読み込みのためだけに開かれた場合，ファイルには何も影響はない．ファイルが書き込みのためにオープンされた場合は，最後にオープンされてから現在オープンされているエクステントに対してなされた変更は無視される．しかし，他のエクステントでなされた変更はたぶん残る．ユーザーが書き込んだデータのいくつかを失いたくなければ，書き込みのためにオープンされたファイルを fabart してはいけない．

```
unsigned cfsize (fd)
```

は，ファイルに関連する r/w ポインターに影響せずに，正確なファイル・サイズ（セクターで）を計算する．ここで戻されたサイズは，ファイル・サイズを計算するのに使用される BDOS 機能 (35) とちがって，新しいエクステントがクローズされる前に，そのエクステントへ書かれたデータを戻す．

```
int oflow (fd)
```

与えられたオープン・ファイルに関連する FCB のランダムレコード・フィー

ルドの高位（3番目）のバイトからオーバーフローが検出されたなら，真（0以外）を戻す．

int errno（ ）

は，ファイルI/O操作の最後に検知されたエラー状態のコード・ナンバーを戻す．次のコードに関係するエラーメッセージのリストを参照せよ．

char * errmsg (errnum)

この関数は，errnoによって戻されたエラー・レコードを基に，対応するエラー・メッセージのポインタを戻す．

0	まだエラーは起こっていない	No error occured yet
1	書かれていないデータの読み取り	Reading unwritten data
2	ディスクの容量不足	Disk Out of space
3	現在のエクステントを閉じることができない	Can't close current extent
4	書かれていないエクステントへシークした	Seek to unwritten extent
5	新しいエクステントをつくることができない	Can't create new extent
6	ディスクの末尾を過ぎてシークした	Seek past end of disk
7	悪いファイル記述子が与えられた	Bad file descriptor given
8	ファイルが読み込みのために開かれていない	File not open for read
9	ファイルが書き込みのために開かれていない	File not open for write
10	ファイル記述子の領域が残っていない	No file descriptor slot left
11	ファイルがみつからない	File not found
12	openに悪いモードが与えられている	Bad mode given to open
13	ファイルをつくることができない	Can't cre ate file
14	65535番目のレコードを過ぎてシークした	Seek past 65535th record

in setfcb (fcbaddr, filename)
char fcbaddr [36] ;

```
char * filename;
```

filename によって示された0で終結する名前を用いて，アドレス fcbaddr で配置レコード36バイトの CP/M ファイル・コントロールブロックを初期化する．ファイルネーム文字列の中の小文字は大文字へ変換され，適切な数の空白は，ファイルネームと fcb の拡張フィールドの両方を埋め込むために生成される．fcb の次のレコードとエクステント・ナンバーのフィールドは0にセットされる．

ファイルネームの文字列の中で奇妙な記号（名前やファイル・コントロール，ブロックの拡張フィールドの中では，たいてい望ましくない種の）に出会うと，そのときは違反記号とそのファイルネームの残りは無視される．

```
char * fcbaddr (fd)
```

は，記述子 fd を持つオープン・ファイルと関連している内部の（たいてい目にみえない）ファイル・コントロール・ブロックのアドレスを戻す．fd がオープン・ファイルのファイル記述子でない場合，－1が戻される．

3.4.5 バッファ付きファイル I/O 関数

```
int fopen (filename, iobuf)
char * filename;
struct _buf * iobuf;
```

は，バッファ付き（1回につき1つの情報）入力のために指定されたファイルをオープンする．そして，iobuf によって示されたバッファを初期化する．Iobuf は，バッファされた I/O ルーチンによる使用のために予約される BUFSIZ の値は BDS C の標準 I/O ヘッダーファイル (BDSCIO. H) によって決定される．バッファ付き I/O を使用しているバッファの構造体は，

```
struct _ buf {
        int _ fd;
        int _ nleft;
        char * _ nextp;
        char _ flags;
        char _ buff [NSECTS * SECSIZ] ;
};
```

しかし，ユーザーにとって本当に問題なのは，それがBUFSIZ-バイト領域で

```
char samplebuf [BUFSIZ] ;
```

によって宣言されうる．ということである．

リターン値は，オープンされたファイルのためのファイル記述子の番号である．これは，最初のエラーのテストの後にセーブされる必要はない．というのは，ファイル記述子の値が他のバッファ付きI/O関数による使用のために，I/Oバッファの中で自動的に維持されているからである．エラーのときは－1が戻される．

```
int getc (iobuf)
struct _buf * iobuf;
```

は，fopenによって開かれたiobufでバッファを持つバッファずみの入力ファイルから次のバイトを戻す．特別なコードは認識されない； コントロールZはコントロールZ（－1でなく）として通る．CRとLFは普通の記号である．コンソールかリーダーから入力を行うためには，バッファされた入力機能を持つiobuf引数のかわりに0か3を使用する．"getc (0)"は"getchar( )"と等しい，"getc (3)"は，CP/Mの"リーダー"から文字を読み取る．エラーのとき，あるいは物理的なファイルの終わりのときに－1が戻される．getcでテキスト・ファイルを読み取っているとき，値0xla (CPMEOF)と普通の物理的なフ

ァイルの終わりの値(－1，かERROR)の両方は，ファイルの終わりを示すものとしてみなされるべきである．というのは，あるCP/Mのテキストエディタはある状況のもとでは，テキスト・ファイルの末尾に0xla（コントロールZ, CPMEOF）バイトを書かないからである．

```
ungetc (c, iobuf)
char c;
struct _buf * iobuf;
```

は，iobufの入力バッファへ文字cを押し戻す．同じファイルでの次のgetcの呼び出しはcを戻す．一度に一つ以上の記号は戻せない．

```
int getw (iobuf)
struct _buf * iobuf;
```

getcへの2回の連続した呼び出しによって，iobufのところで，バッファを持つバッファ済みの入力ファイルから，次の16ビットのワードを戻す．エラーのとき－1が戻される．

```
int fcreat (filename, iobuf)
char * filename;
struct _buf * iobuf;
```

は，filenameで指定されたファイル（同じ名前によって既存のファイルをまず削除して）をつくる．そして，バッファ付き出力のためにファイルをオープンする．IobufはBUFSIZE-バイトのバッファをさし示すべきである．そのファイルのためのファイル記述子fdを戻し，エラーのときは－1を戻す．

```
int putc (c, iobuf)
```

```
char c;
struct _buf * iobuf;
```

は, iobufのところでバッファを持つ, バッファ付き出力ファイルへバイトcを書き込む. 翻訳はなされない； テキストの行はCR-LFの組み合わせ（標準CP/Mソフトウェアとの適合性のために), あるいは, UNIXニューライン記号(効率性と率直さを増すために)のいずれかによって区別させることができる. iobufのかわりに使用できる1から4までの値は, ファイルに対する出力のかわりに, 標準出力, リスト装置, パンチ装置, 標準エラー（コンソール）装置に対して出力する.

"putc (c, 1)"は, "putchar (c)"と等しい.

"putc (c, 2)"は, 文字をCP/M"リスト"装置へ書き込む.

"putc (c, 3)"は, 文字をCP/M"パンチ"装置へ書き込む.

"putc (c, 4)"は, 文字を標準エラー・ストリームに書き込む. これはCP/Mのもとではいつもコンソール出力である. これは, I/Oダイレクションパッケージ(DIO)が使用され, 標準出力がファイルへ命令されるような応用の中で, 出力がコンソールへ行くことを保証するために使用される. ファイルへテキストを書き出した後, 必ずコントロールZ (0x1a, COMEOF) バイトを用いてテキストを終結させる. エラーのときは－1を戻す.

```
int putw (w, iobuf)
struct _buf * iobuf;
```

は, putcへの2回の連続した呼び出しによって, iobufのところでバッファを持つバッファ付きの出力ファイルへ, 16ビット・ワード<u>w</u>を書き込む. エラーのとき, －1を戻す.

```
int fflush (iobuf)
struct _buf * iobuf;
```

は，出力バッファ iobuf をフラッシュする．これは出力バッファが，最後に埋まってから出力バッファに書き込まれた文字が，ディスク上のファイルへ書き込まれたことを確認する（与えられたプログラムは exit ルーチンが，すべてのファイルを閉じる前に中断しない）．Fflush はバッファされた Output ファイル（出力）を用いた使用のためのもので，入力ファイルでそれを使用しようと試みても効果はない．

出力ファイルがクローズされるとき（fclose 関数によって）だけでなく，出力バッファがいっぱいになったときは，いつでも自動的に fflush が起こる．

```
int fclose (iobuf)
struct _buf * iobuf;
```

指定されたバッファ付き I/O ファイル（(fopen による）読み取りか，あるいは（fcreat による）書き込みのいずれかのためにオープンされた）をクローズする．ファイルが書き込みのために開かれたのなら，ファイルを閉じる前に自動的に fflush が出力バッファをフラッシュするために，呼び出される．

注意：　バッファされた出力ファイルに対して書かれたテキストを持つバッファ付きの出力ファイルをクローズする前に，必ず，CP/M の“テキスト・ファイルの終わり”のマーカー（CPMEOF）をファイルへ書き込むこと．

```
int fprintf (iobuf, format, arg1, arg2, …)
struct _buf * iobuf;
char * format;
```

書式付き出力が，コンソールに対するかわりに，バッファ付き出力ファイルに対して書かれることを除けば，printf と同様である．エラーのときは－1を戻す．

```
int fscanf (iobuf, format, arg1, arg2, …)
struct _buf * iobuf;
char * format;
```

テキスト入力が，コンソールからのかわりに，入力バッファ iobuf から走査されることを除けば，scanf と同様である．fscanf の今あるバージョンは，データの各行は完全に走査されることを求める．すべてのフォーマット仕様が満された後に，ファイルから読まれた行の中の項目は捨てられる．うまく割り当てられた項目の数を戻す．あるいは，ファイルを読んでいるときにエラーが起これば，－1を戻す．

```
char * fgets (str, iobuf)
char * str;
struct _buf * iobuf;
```

指定されたバッファ付きの入力ファイルから行を読み込み，str によってさし示された位置からのメモリーの中にそれをおく．これは，CP/M の規則がテキスト・ラインの末尾に CR（キャリッジ・リターン）と LF（ニューライン）の両方を持っているために扱いにくいものである．fgets は，Cプログラムから処理する．テキストをより簡単にするために，ファイルに含まれている CR-LF の組み合わせから CR を自動的に取り除く．LF の後の CR 記号は手をつけずにおかれる．LF は，文字列の一部分として含まれる．そして，0が後につづく．読み込まれるラインの長さに関してはチェックはない．考えられる最長のラインを入れるために，str のところで十分な余裕があるかどうか確認するのに，十分な注意が必要である（ラインは完全であるとみなされる前に，ニューライン記号によって終結されなければならない）．EOF のとき，ゼロが戻される．EOF が物理的な EOF（ファイルの最後のセクタを通りすぎて読み取ろうと試みる）であっても，あるいは，ファイル中のコントロールZ（CPMEOF）記号でも，ゼロが戻される．さもなくば，文字列に対するポインタ（パラメタ str と同様の）

が戻される.

```
int fputs (str, iobuf)
char * str;
struct _buf * iobuf;
```

は，strのところのメモリーからの0で区切られた文字列を指定されたバッファ付きの出力ファイルへ書き込む．CP/Mがうまく実行されるために，ニューライン記号はCR-LF組合せへ変換される．ニューラインの前に，空白(ゼロバイト)が文字列の中ではつけられる場合，出力時のラインに付加されたライン・ターミネイターはないであろう（部分的なラインを書くことが可能).

3.5 DMAビデオ・ボードのための作図関数

```
setplot (base, xsize, ysize)
```

は，プロセッサ・テクノロジー（R.I.P）のVDM-1のように，メモリー・マップ化された"DMA"ビデオ・ボードの物理的特徴（スターティング・アドレス，寸法)を定義する．Baseは，ビデオ・メモリーの開始アドレスである．xsizeは，ディスプレイのラインの数である．ysizeは，行の数である．Setplotは，プログラム実行のスタート時に，一度呼び出すだけでよい．それから関数clrplot, plot, txtplot, lineは与えられたパラメタを得て機能する．

```
clrplot ( )
```

は，メモリー・マップされたビデオ・スクリーンをクリアする（空白でうめる).

```
plot (x, y, chr)
char * chr;
```

は，ビデオ・スクリーン上の座標（x, y）に文字chrを書く．（x, y）は，down,

across と読まれる．ここで

```
0 <=x <xsize,
0 <=y <ysize,
```

の値でなければならない．

```
txtplot (string, x, y, ropt)
char * string;
```

は，スクリーン上の（x, y）の位置から，ASCII 文字列を書く．ropt がゼロ以外のものなら，文字列の各バイトは，表示される前に値 0x80 でもって，論理的に OR される．これは高位ビットを 1 にすることにより，(VDM-1 のようなボードでの）反転文字が出力されるようにする．あるいは，他のいくつかのボードでは，おかしなでたらめな表示をする．

```
line (c, x1, y1, x2, y2)
```

この関数は，ポイント（x1, y1）と（x2, y2）の間に，曲線（64×16の画面サイズでは，まっすぐにみえる線をつくる方法はないからである！）を引く．

そのラインは，記号 c でできている．Line は，64×16の画面サイズでしか作動しない．

第 4 章

" The C programming Language "の付録Aに関する註釈

4.1 序

BDS Cは，UNIXのサブセットである．ゆえに，Cリファレンス・マニュアルのほとんどの部分は，BDS Cに直接適応する．この付録の目的は，BDS Cがその文書に従わないセクションに註釈をつけることである．

2つのコンパイラ間の相違について，一般的に要約した後，その文書からの適切なセクションのナンバーを参照し，また，BDS Cがそこで説明されていることとどう違うのかを説明することによって(BDS Cに関して)より詳しく述べる．

次にあげるのは，UNIX CとBDS Cの最も重要な偏差の要約である．

1. 全ソース・ファイルは，ウィンドウを通してパスされるかわりに，すぐにメモリーへロードされる．これは，有効なメモリーサイズに対して単一ソース・ファイルの最大の長さを制限する．
2. コンパイラは，中間アセンブリ言語ファイルを作らずに，8080マシーン・コードを直接生成する．
3. BDS Cは，C言語ではなく，8080アセンブラ言語で書かれている．BDS Cがそれ自身を用いて書かれた場合，そのコンパイラは，何倍も大きくなる

だろうし，現在のスピードと同じ速さで実行しない．我々はここで8080コードを取り扱っているのであって，UNIXのようにPDP-11コードではないことを憶えておいてほしい．

4. 変数タイプ，short int, long int, float, double はサポートされない．
5. 明白に宣言できる記憶クラスはない．static と resister の変数は存在しない．すべての変数は，external か automatic かのいずれかであり，それらは宣言された文脈によって決まる．
6. 宣言の複雑さは，あるルールによって規制される．
7. 初期化は，サポートされない．
8. 文字列の記憶割り当ては，明白に操作されなければならない（自動割り当て／不要部分の整理のメカニズムはない）．

4.2 付録Aに対する註釈

次は，Cリファレンス・マニュアル[16]に対するセクションごとの註釈である．簡潔にするために，上にあげたいくつかの項目は再び示されない．本の中でみつけられる．float 浮動小数点，long 倍精度整数，static 静的変数，initialization 初期化などに対する参照は無視されるべきである．

1. はじめに

BDS C は，CP/M OS を備えた8080マイクロコンピュータシステムのためにデザインされていて，与えられたCソース・プログラムから直接8080二進のマシーン・コードを（特別な再配置可能フォーマットで）生成する．当然 BDS C は，Z80 あるいは8085のように，8080とかなり適合するプロセッサでも実行されうる．

16) "C programming Language" の本の付録A

2.1　コメント（註釈）

コメントは，デフォルトで入れ子が可能である．BDS Cのプロセッサのコメントを UNIX Cが行う方法にするために，－cオプションをコンパイラ CCへ与えられなければならない．

2.2　識別子（名前）

大文字，小文字は変数，構造体，共有体，配列名では区別されている（異なる）が，関数名はこの限りではない[17]．関数名は混乱をさけるために，単一の文字型（大文字か小文字のいずれか）で書かれるべきである．たとえば命令文

```
char foo, Foo, FoO;
```

は，異なる名前を持つ3つの文字型変数を宣言する．しかし，2つの式

```
printf ( " This is a test " );と
prINTf ( " This is a test " );
```

は，等しい．

2.3　キーワード

BDS Cのキーワード：

```
int         else
char        for
struct      do
union       while
unsigned    switch
```

17) 関数名は，内部で大文字として記憶される．

```
goto          case
return        default
break         sizeof
continue      begin
if            end
register      void
```

キーワードの場合，大文字，小文字の区別は無視される．たとえば，WHILEはwhileに等しい．

キーワードは，識別子（たとえばcharflag）の中に含まれることはできるが，キーワードと同じ名前を持つ識別子は認められない．

左右のカーリー・ブレイブス記号｛と｝をサポートしない端末装置では，キーワードbeginとendが取ってかわる．これらの識別子を持つことはできない．というのは，これらはコンパイラによってキーワードと認識されるからである．

4. 名前には何があるか？

externalと，automaticという2つの記憶クラスがあるだけだが，それらは明白に宣言されない．識別子が宣言されているような文脈は，いつもその識別子が外部のものであるか，あるいは自動的であるかどうかを決定するのに，十分な情報を与える．関数の定義の外側に現われる宣言は明らかに外部のものであり，関数定義の範囲内でのすべての変数の宣言は自動的である．

自動変数は，宣言のポイントから現在の関数定義の末尾まで，語意の有効範囲を持っている．単一の識別子は，与えられた関数内で2回以上宣言されることはない．これが意味するのは，局所の構造体メンバーや構造体タグは，局所変数と同じ名前（またはその逆）を持つことができない．特別なケースについてはサブセクション11.1をみなさい．

BDS Cでは，関数の中にblocksの概念はない．複合文の最初に，局所変数を定義することができるが，それは後に宣言された局所自動変数と同じ名前を

持つことはない．また，その語意の有効範囲は，複合文の末尾を通りすぎて，関数の末尾に至るまでずっと続く．

すべての自動変数宣言は，関数定義の始まりに制限され，他の複合文の先頭で変数を宣言する実行は避けられるようにすることを強くすすめる．

いくつかのファイルが，1つのセットになった共通外部変数を共有している場合，すべての外部変数宣言は，関連する[18]ファイルの各々の範囲内で，等しく命令されなければならない．BDS C における外部変数のメカニズムは FORTRAN のラベル無し COMMOM 文と，ほとんど同じように操作される．たとえば，他のファイルがa，b，cだけを使っていて，ユーザーの main ソース・ファイルが外部変数a，b，c，d，eを順に宣言する場合，2番目のファイルは，d，eを宣言する必要はない．一方，2番目のファイルが，a，b，cでなくて，aとeを使用した場合，dとe（2番目のファイルから）がaとb（最初のファイルから）とオーバーラップしないように，また，大きなトラブルを起こさないように，変数のすべてが宣言されなければならない．加えて不便なことに，そのソース・ファイルがそれらすべての外部変数を使っているかいないかにかかわらず，"main"関数を含むソース・ファイルの範囲内で宣言されなければならない．

すべての共通外部宣言が，単一の"H"ファイルの中でキープされる限り，また# include が"H"ファイルを読み取るように，プログラムの各ソース・ファイル内で使用される限り，トラブルが起こることはない．いずれにしても比較的少なければならない．

6.1　文字と整数

BDS Cでは，符号拡張は行われない．文字は0～255の範囲で8ビットの符号なしの値として解釈される．

---

18) このような場合に，すすめられる方法は，すべての共通外部変数宣言を含む単一ファイルを準備することである．そのファイルは，拡張名.H（"ヘッダー"）を持っているべきで，各ソース・ファイルの先頭で# include プリプロセッサを使用して,指示されるべきである．

BDS Cでは，CHAR変数は負の値を持つことはできない．

get cのような関数のリターン値をテストするは，注意しなさい．これは，エラーのときは普通“文字”でなくて-1を戻す．実際，リターン値はget cのリターン値を文字変数へ割り当てる場合，－1の値は8ビットの記憶セルでは255へとかわる．そして，－1と等しいかをみるために，文字をテストしても真を戻さない．こういった状況のときには注意しなさい．

文字でのほとんどの演算は，その文字を高位バイトに0を持つ16ビットの数へと変換することによって達成される．割り当て式のように，非演算操作においては，バイトだけを基準にしているchar値を処理することによって，コード生成を最適化する．これを利用するために，char変数として0～255の範囲でとどまると，ユーザーが信用している変数を宣言しなさい．

7. 式

ゼロによる除算とゼロによるmodは，両方ともゼロの値になる．この場合，どんなエラーも発生しない．

7.1 基本式

関数呼び出しの中のパラメタの評価の順は，逆になる．たとえば，最後のパラメタが，最初に評価され，スタック上でプッシュされる．それから，最後から2番目のが評価され，スタック上にプッシュされる，というように．これは，パラメタが呼ばれている関数に対して上向きの順で現われるように行われる．また，パラメタの有効数を関数が取るようにするためもである．

7.2 単項演算子

演算子

(型指定子) 式

sizeof (型指定子)

は，実施されない．sizeof演算子は式が配列でないように与えられた

sizeof 式

というフォームの中で使用されるsizeofを配列と取るには，その配列はそのsizeofを取られる構造体として認めて，その配列自身によって，構造体の中へすべて配置されなければならない．もう一つの可能性は，sizeofを配列の中の単一エレメントとしてとることである．それから，それを配列全部のサイズを生み出すために，配列の中のエレメントの数で掛けることである．

7.5　シフト演算子

操作　>>　はいつも論理的（0で埋める）である．

7.11.7.12　論理的 AND と OR 演算子

BDS Cでは2つの演算子&&と｜｜は，等しい優先順位を持ち，それがUNIX Cのもとでないような場合には，かっこを必要とする．&&と｜｜の複雑な組み合わせを含む式は，いずれにしても基本的には，混み入ったものになる．また一般的な主義として，かっこを使用するべきである．

8.　宣言

宣言は，フォーム：

　　型指定子　　　宣言リスト

を持つ．“記憶クラス”指定詞はない．

8.1　記憶クラス指定詞

実施されない．

8.2　型指定詞

型指定子：

　　char

　　int

unsigned

resister

struct または union 指定子

resister 型は，変更子（たとえば，register unsigned foo;）として使用されないならば，int と同義語と思われる．この場合，完全にそれは無視される．

キーワード void は，int と同義語として扱われ，関数が値を戻さない事実を文書証明するために使用される．他に認められている“形容詞”はない．

unsigned int foo;は，

unsigned foo;

として書かれなければならない．

8.3 宣言子

初期化は認められない．ゆえに，宣言子リストの構文は次のようになる．

宣言リスト：

宣言子

宣言子，宣言子リスト

8.4 宣言子の意味

UNIX C は，任意の複雑な型の組み合わせを認められているし，

```
struct foo *(* (* bar [3] [3] [3] ( )) ( );
```

ここでは，bar 3 × 3 × 3 の配列のポインタが関数のアドレスをさし，その関数の返すポインタが関数のアドレスを示し，それが返すポインタタイプ foo の構造をさしている．

いずれにせよ，BDS C はそういった特殊な宣言を認めないだろう．

まず，単純型指定が

char
int
unsigned　struct
union

によって定義されるようにし，それから，

スカラ型指定：

simple-type
pointer-to-scalar-type
pointer-to-function

によって，スカラ型が定義されるようにすること．

確定的なスカラ型である関数へのポインタはそれに割り当てられた関数のアドレスを持ち，その関数を（正しい構文を用いて）コールするために使用される変数である．BDS C がこういったものを内部で操作する方法のために，関数へのポインタの変数に対するポインタは，正しく作動しない．しかし，他のスカラ型（構造体，共有体，関数へのポインタを除く）を戻す関数に対するポインタはOKである．

今までのところ，スカラ型は，

```
int x, y;
char * x;
unsigned * fraz;
char ** argv;
struct foobar * zot, bar;
int *(* ihtfp) ( );
```

といった宣言をカバーする．

（上記の例の最後のものは，ihtfp ポインタを，整数を戻す関数に対するポインタであるように宣言する．）

スカラ型の概念をつくり，我々は配列がスカラ型の目的の一次元あるいは二次元のコレクション（関数へのポインタの変数を含む）であるように定義する．今，我々は，

```
char * x[5][10] ;
int ** foo[10] ;
struct steph bar[20][8] ;
union joyce * ohboy[747] ;
int *(foobar[10] )( );
```

といった構成を持ちうる．

（上記の例の最後のものは，foobarが整数を戻す関数に対する10のポインタをつくり上げている配列であるように宣言する．）

次に，我々は関数が関数へのポインタ，構造体，共有体以外のスカラ型を戻すことを認めている（構造体と共有体に対するポインタは除外しない）．

もっと例をあげると，

```
char * bar( );
```

は，barが文字ポインタを戻す関数であることを宣言する．

```
char *(* bar)( );
```

は，barがポインタを戻す関数に対するポインタになることを宣言する．

```
char *(* bar[3][2] )( );
```

は，barが文字ポインタを戻す関数に対する個々のポインタの3×2の配列であることを宣言する．

```
struct foo aot( );
```

は，aotがタイプfooの構造体を戻す関数であるように宣言しようとする．関数

は構造体を戻すことはできないので，これは予知できない結果を生むだろう．

```
struct * foo zot ( );
```

は，OKである．aotは，タイプfooの構造体のポインタを戻すものとして宣言される．

BDS Cの1つの重大は"悪い特徴"は，明白な配列へのポインタは宣言することができないことである．いいかえれば，

```
char (* foo) [5] ;
```

のような宣言は，fooが配列に対するポインタになることを宣言するときはうまくいかない．BDS Cコンパイラ（そしてそのプログラマーも）が比較的単純なところがあるために，先行宣言は，

```
char * foo [5] ;
```

と同じ意味を持って終わる．

より楽観してみると，配列として宣言された関数のパラメタは，"配列へのポインタ"として内部で操作され，適切な配列識別子が式の中で使用されるときはいつでも自動的な関数的手段が取られるといったことをひき起こす．これは，関数に対してpiと同じ位かんたんにパスする配列をつくる．このメカニズムの多方面にわたる例のために，BDS Cパッケージのいくつかのバージョンに含まれるOthelloプログラムを調べなさい（Cユーザーズ・グループでいつも手に入る）．

8.5 構造体と共有体の宣言

"ビット・フィールド"は実施されない．我々が持っているのは，次のとおりである．

struct-or-union-specifier:

　　　　struct-or-union {struct-decl-list}

struct-or-union identifier {struct-decl-list}
struct-or-union identifier

struct-or-union

`struct`
`union`

struct-decl-list:

struct-declaration
struct-declaration struct-decl-list

struct-declaration:

type-specifier declarator-list;

declarator-list:

declarator
declartor, declarator-list

構造体定義の中のメンバーとタグの名前は，他の局所識別子の名前と同じであってはならない．(一つの関数につき)一つ以上の構造体あるいは共有体がメンバーとして与えられた識別子を使用できるのは，すべての場合が同じタイプとオフセットを持つときだけである．サブセクション11.1をみなさい．

8.6 初期化

申し訳ないが，初期化は認められていない．

すべての外部変数は，自動的にゼロへ初期化される．(これは，1.50以前のコンパイラの場合は，そうでないことに注意しなさい．)

8.7.8.8 型名，Typedef

BDS Cに適応しない．typedefは実施されない．

9.2 複文，すなわちブロック

BDS Cにはブロックはない．変数はブロックに対してローカルなように宣言されない．関数の中のどこででも現われる宣言は，その関数の終わりまでその効力は残る．

9.6 For文

本書は少しごたごたしている（本書がユーザーを混乱させてなかったら，次にはっきり述べていることは，きっといいでしょう…）

for命令文は，説明されたように完全にwhile命令文と等しくはない．というのは，continue命令文は，forループの命令文の部分を実行している間に出会うすべてであるなら，コントロールは式3にパスするだろう．だがwhileバージョンにおいては，continueはコントロールが直接ループのテスト部分にパスするようにする．また，その特殊な繰り返しの間は，式3を実行しない．一方，セクション9.9で与えられた表示は正しい．なぜならば，増加は明白に書かれるのでなくて，暗にほのめかされる（contin:で起こると）からである．

これは単なる文書提示における矛盾である．UNIX Cコンパイラ（私の知る限りでは）とBDS Cコンパイラの両方がforのケースを正しく操作する．

9.7 switch文

一つのスウィッチ構成につき，200以上のcase文を指定できない．

複数のcase文は各々，1として数えることに注意しなさい．だから命令文，

```
case 'a': case 'b': case 'c': printf(" a or b or c")
```

は，3つのcaseと数える．

9.12 名札付き文

case あるいは default に直接つづくラベルは認められない．そのラベルは最初に書かれるべきであり，それから case か default のキーワードによって続けられる．例えば，

```
case 'x' : mumble:zap=frltz;
```

は正しくない．

```
munble:case 'x':zap=frotz;
```

へ変更されるべきである．

10 外部定義

型指定子は，関数定義（デフォルトは int である）を除いたすべての場合に明白に与えられなければならない．

11.1 構文範囲

構造体や共有体の範囲内でのメンバーとタグは，宣言された識別子の他のタイプに等しい名前を与えられるべきでない．BDS C は単一識別子が 1 回に一つ以上のことに使用されるのを認めない（局所識別子が類似した名のついた外部識別子を一時的におおっているときを除く）．これが意味しているのは，

```
struct foo {            /* define struct of type "foo" */
    int a;
    char b;
} foo [10] ;            /* define array named "foo" made up
                           of structures of type "foo" */
```

のような宣言を書くことはできないということである．これは，たとえUNIX Cがこういったものを許しているとしても，結果的には混乱するし，いずれにしても使用されるべきでない．

このルールには，1つの例外があり，構造体メンバーを含んでいる．コンパイラは，1)タイプと2)記憶オフセット（その構造体がベースから）が両方の場合で，同一である限り異なる．構造体の定義の範囲でメンバーと同じように，使用されている識別子を黙認する．たとえば，次のものはこの許容されている方法で識別子"cptr"を使用する．

```
struct foo {
    int a;
    char b;
    char * cptr;        /* type: char *, offset: 3 */
};
struct bar {
    unsigned aa;
    char xyz;
    char * cptr;        /* type: char *,  offset:3 */
};
```

11.2　外部定義の範囲

externのキーワードはない．すべての外部変数は，ファイルのサブセットを使用する．各ファイルの範囲内で同じ順で正しく宣言されなければならない．また，プログラムの中で使用されるすべての外部変数は，"main"関数を含むソースファイルの範囲内で宣言されなければならない．

次にあげるのは，外部定義が普通どのようにして操作されるかである：ランタイム・パッケージのロケーション0015h（たいていメモリー・ロケーション0115h)は外部変数領域のベースに対するポインタを含んでいる．すべての外部

変数は，このポインタ[19]に索引をつけることによってアクセスされる．全プログラムのための外部データ領域は，“main”ソース・ファイルで定義されたすべての外部データによって必要とされるスペースに等しいようにCLINKによって仮定される．外部記憶あるいは，外部名に関してCRLファイルの範囲内では，情報は記録されないので（関連するバイトのトータル数以外，そして任意に外部の明白なスタート・アドレス），各ソース・ファイルが，外部宣言の同一のリストを含んでいることを確認するのは，ユーザーの責任になる．名前は，別々のファイルの中で各々符号する外部変数のために，必ず同一である必要はないけれど，型と記憶クラスは，混同をさけるために必ず一致しているべきである．

BDS Cの外部変数を，一つの大きなFORTRANのようなCOMMONブロックとみなしても，見当はずれということもないだろう．

注意： ライブラリ関数allocとfreeを使用する場合，ユーザーはプログラムの中にヘッダーファイルBDSCIO.Hを含んでなければならない．というのは，BDSCIO.Hの中で宣言されたallocとfreeの必要とする外部データがいくつかなければならないからである．また，外部変数を持つソース・ファイルの範囲内でこういった宣言を省くと，望ましくないデータ・オーバーラップを引き起こす．

12.1 綴りの置換

#defineプリプロセッサの命令のすべてのフォームはサポートされている．パラメタ化された定義を含んでいる．再帰的（互いに参照する）にパラメタ化された#define操作は検知されない．もしそれをやってみると，ストリングのオーバー・フローを引き起こす．

---

19) CCに対する －e xxxx オプションは，絶対的なロケーション xxxx で外部変数領域を位置指定するために使用される．それによって，かなりスピードがアップし，コンパイラのつくり出すコードを短かくする．それでもすべての宣言制約条件は，依然として守られなければならない．

12.2 ファイルの包含

二重引用符が，ファイルネームを区切るために使用されていて（たとえば，#include " filename "のように），明白なドライブあるいは，ユーザー領域の指示子が，ファイルネームに先行して現われた場合，そのファイルは現在のディレクトリだけの中で常駐していると思われ，ファイルがそこになければコンパイル作業は打ち切られるだろう．アングル・ブラケット（#include <filename>のように）が使用されている場合，デフォルトのディスクのドライブユーザー領域のみ（第1章でのべられたように）が検索される．

含まれているファイルがすでにロードされてしまった後に，条件つきのコンパイル命令が後のパスで処理されるだけである．一方#include 命令は，ソース・ファイルがディスクから読まれるときに，オン・ザ・フライで処理されるということに注意しなさい．それゆえに，コンパイラは，その条件を偽と評価したときでも，条件つきコンパイル・ブロックの範囲内で配置された#include 命令を処理しようとする．

すべての#include 命令がみつかる限り，事は正しく運ぶ．というのは，その条件が処理されるとき，適切なコードはただ後になって無視されるだろうから．しかし，もし，#include 命令によって指定されたファイルがみつからない場合，CC はエラーをプリントし，コンパイルを打ち切る．

ファイル包含は，適度な深さに入れ子にされうるが，エラー・リポートは，ネスティングのレベルだけを認識する．何が起こるかを正しく知るために，包含ネスティングのレベルを変更し，CC の" －p "オプションを用いて，実験してみるとよい．

12.3 条件付きコンパイル

すべての標準条件付きコンパイル命令は，サポートされるが，#if <expr> 命令によって取られた式は次の構文に限られる：

```
〈式〉: = 〈式2〉                              または,
           〈式2〉  &&  〈式〉                 または,
           〈式2〉  ||  〈式〉
〈式2〉: = 〈10進定数〉                        または,
            ! 〈式2〉                         または,
            (〈式〉)
```

〈10進定数〉は，シンボリックである(# define の置換が完成した後に，平易な10進定数を産む)．しかし，# if プロセッサによって，論理的値として，いつも取り扱われる．つまり，0の値は偽であり，その他の値は真である．条件付きコンパイル命令のネスティングは，十分サポートされている．

12.4　行制御

実施されない．

15.0　定数式

BDS C は，定数式が次のキーワード，左スクエアー・ブラケット([ )，CASE キーワード，割り当て演算(=)，コンマ，左ペアレンセス( ])，return キーワードのうちの一つのすぐ後に現われるときだけ，コンパイル時に定数式を単純化する．これらのキーワードのうちの一つにつづかない定数式は，コンパイル時に単純化されないことを保証されている．

定数式のコンパイル時の評価を確実にするための標準的な手順は，定数以外のエレメントを含むより大きな式の中に含まれているとき，特にかっこの中に定数式を置くことである．このように，

```
x = x + y + 15 * 10;
```

のような命令文は単純化されない(これは10と15を掛けるためのコードをコンパイルに生成させる)．また，一般的に，

```
x = x + y + (5 * 10);
```

というようなより良いフォーム,よりも長くてスローなコードを生む.

定数と定数式でのすべての乗算計算は，符号なしのオペレーションとしてなされる.

18.1 式

単純演算子は，次のとおりである.

```
*  &  -  !  ~  ++  --  sizeof
```

二項演算子 &&と | | は同じ優先度を持つ.

sizeof 演算子は，配列のサイズを正しく評価できない.

18.2 宣言

宣言のための安全な構文は,

```
data-definition:
  type-specifier declarator-list;
```

18.4 外部定義

データ定義：

```
type-specifier declarator-list;
```

18.5 プリプロセッサ

次のプリプロセッサ命令が，サポートされる.

```
# define identifer token-string
# include " filename"
```

```
# inclure 〈filename〉
# if expression
# ifdef identifier
# ifndef identifier
# else
#endif
# undef identifier
```

#define は，ソース・ファイルのどこにでも現われうる．ファイルの最後まで，あるいはその識別子が再び#define されるか．undef されるかするまで，その有効範囲は広がる．

```
# if 〈expr〉
```

の命令は，サポートされる．しかし，正しい式エレメントは定数（記号定数を含む）と小さなセットになった演算子に限られる．#if 命令はユーザーが，注釈付きの，また注釈なしの#difine 命令を持つプレイゲーム　そして#ifdef/#ifndef もしくはその一方を使用することに頼らずに，システム従属の条件付き式を書くことを認める．完全な構文については，上のセクション12.3をみなさい．

#include 命令は，条件付きコンパイルの内側に現われてはならない．これは，#include 命令は，入力ファイルがディスクから読まれるときに，コンパイラによって，オン・ザ・フライをすべて処理されるかたちで，条件付きコンパイルの処理は，全ファイルが読み込まれてしまった後になるまで起こらないからである．このように，たとえ#include 命令が偽りの条件付きコンパイルの中で置かれるとしても，#include 命令はいつもコンパイラに指定されたファイルを読み取らせようとする．これは，設計の欠陥のように思われるかもしなないが，すべての条件付き命令を処理し，標準8″単密度CP/Mディスクから，適度なスピードでソース・ファイルを読み込む方法は他にない．

条件付きコンパイルを使用するとき，各々の#else 命令は，符号する#endif 命令によってつづけられなければならないことに注意しなさい．

ファイル包含は，どんな深さ[20]にも入れ子にするかもしれない．

しかし，－pのCCオプションと，CCとCC2のためのエラー・レポートの両方は，ファイル包含がシングルレベルに限られると，処理するのが簡単になる．

20) だが，互いの包含的ファイルは，きっとオーバーフローを引き起こす．

――付　録――

付録　A

多方面にわたる注意書

— 演算子＝は，割当のためだけに使用される．関係演算子"それと同じ"は，＝＝演算子によって代表される．混同しないように注意すること．まちがったものを使用しても，コンパイラに診断メッセージを生成させないことになる．というのは，出てきた式は，それらがたとえ望ましい効果をもっていなくても構文上正しいからである．

— キーワード begin と end は，左右のカーリー・ブレイス（{と}）にとってかえられる．端末機にカーリー・ブレイス記号を持たないユーザーも，コンパイラを使用できるようにこの特色が与えられている．審美的にいっても，カーリー・ブレイスは begin と end よりもはるかに読みやすいリスティングをつくるし，可能ならいつでも使用されるべきである．

— コンパイラ操作中のエラー回復は，ある場合，特に聡明ではない．CC か CC2 のいずれかが同じラインかあるいは，いくつかのセットになった行のまわりのひとかたまりのエラーメッセージを出すような場合，最初のエラー・メッセージだけが信じられるべきである．そのエラーが修正された後では，残りは消えてしまうだろう．

— エラー・レポートの中で CC2 によって与えられたライン・ナンバーは，必ずしも正確だと保証されていない．CC はたまに，コードの配列直しをする．たとえば，for 命令文の増加部分は，命令文の部分を越えて，物理的にムーブ・ダウンされる．このように，CC が検知するように備えられていな

い．増加部分の中にエラーがあれば，CC2がそれを検知し，まちがってライン・ナンバーをレポートする．for命令文の増加部分を混乱させないようにしなさい．

あるタイプのエラーは，ソースの残りを走査せずに，実行をやめ，すぐにOSへ戻すことがある．これが起こるのは，たとえばペアレンセスの組み合わせのまちがいや，セミコロンがなかったりして，それが回復できないポイントで，コンパイラが混乱するようなときである．正しい句読点があるべき場所について見当をつけるかわりに，それはエラーをすばやく定め，もう一度，トライするために打ち切る．

Cのmain関数へパスされたargc値は，規則によっていつも正で，引数の数＋1に等しい．コマンド行における引数は，いつも文字列で，値ではない．数字コマンド・ラインのパラメタを，変数へ割り当てるのに適切な値に変換するために，atoi関数のようなものが使用される．

“bdos”ライブラリ関数に関する問題は，それがシステム従属であるので，むしろ扱いにくくさせてきた．普通のデジタル・リサーチCP/Mシステムのもとで正しく走っているプログラムは，bdos関数が使用される場合には，MP/MあるいはSDOS(他にどれほど多くのシステムがあるか知らないが）のもとでは走らないかもしれない．この問題の典型的な徴候は，文字出力において，キーボード上の文字は出力の各文字が現われるようにするために，一度打たれる必要があるということである．その問題を理解するために，まずはじめに我々は，CPUレジスターが，OSのBDOSコールの後にどのようにセットされるのかを正確に理解しなければならない．普通のCP/Mの動作（ライブラリ関数bdosがいつも取っている）は，リターン値の低位バイトを含むためのレジスターAとLのためのものであり，リターン値（リターン値が，1バイトだけならそれは0）の高位バイトを含むためのレジスターBとHのためのものである．CP/Mインターフェイス・ガイドは，“すべての場合，リターンのときにはA＝＝L，そしてB＝＝Lである)と言っている．また，私が示したところでは，CP/M1．4あるいは他のシステムがBとAからHとLの中に値を置かなかった場合，その値

がHL（ここでリターン値はいつもCライブラリ関数によって配置されなければならない）の中にあることを確かめるためには，bdos関数に，レジスターAをレジスターLに，レジスターBをレジスターHへコピーさせる．

だが，すべてのシステムが実際にこの規則に従うわけではない．MP/Mにおいて，HとLはいつも正しい値を含んでいるが，Bはそうではない．だからBがHへコピーされるとき，まちがった値が出る．ゆえに，CP/MとMP/Mの両方のもとで，bdosを実行させる方法は，BとAをHとLへコピーするのを中止することであり，値がそのシステムによって，HLの中にも正しく残されると仮定することである．これはV1.45のために行われたので，少なくとも，CP/MとMP/Mは気を付けている．が…

SDOSのもとで（そして，たぶん他のシステムでは)，レジスターAは時々意義のあるリターン値を含む単なるレジスターである．たとえば，ファンクション・コードのコール（コンソールステータスを調べる）からのリターンのときに，B,H,Lレジスタは，すべて不要部分を含んでいるのをみつけられる．だから，こういったときにコピーが行わなければ，リターン値はAからLへといかないし，結果も悪い．しかし，BがLへコピーされて，Aと共にHへコピーされる場合，その結果はやはり悪い．というのは，Bが不安部分を含んでいるからである．明らかに，SDOSのもとでファンクション・コードを正しく作動させるための唯一の方法は，bdos関数に，レジスタAをLにコピーさせることであり，リターンする前にHレジスタをゼロにしてしまうことである．しかし，Hの中に値を戻す多くの他のシステム・コードはもはや作動しないだろう．そして，それが，問題なのである．いつも，あるシステムを気に入るかもしれないが，すべてのシステムがいつも唯一の標準のbdos関数であるというわけではない．

bdosは，CP/MとMP/Mと共に作動するので，V1.5のためにbdosを残す方法はある（たとえば，レジスターのコピーが全然行われなくて，…HLは正しい値を含んでいると仮定される)．もちろん，これはSDOSやたぶん他のシステムでのすべての場合に作動しない．このような場合，BDOSコールをなすために，callとcallaのいずれかを使用する必要がある．ある

いは，ユーザーのシステムのために正しいレジスターの操作順を実行するためにbdos関数（CASMを使用して）のアセンブリ・コード化された自分自身のバージョンをつくる必要がある．すべての可能なリターン値のレジスター構成をカバーするために，このような関数が一つ以上必要であることに注意しなさい．

よく設計されたCプログラムというものは，ユーザーにコマンド行構文を表示し，打ち切ることによってコマンド行エラーをいつも診断するべきである．これは，一般に"Usage"メッセージとして知られる：　これは，何がコマンド行で予想されるかをユーザーに思いつかせるし，しばしば，何回もそのプログラムを使用する人を助ける．コマンド行・オプションがあれば，それらは，スクェア・ブラケット'［'で示されるべきである．良い見本として，サンプルのコマンド行と共にすべてのオプションの詳細は説明を含むものである．

外部初期化は，コンパイラによってサポートされないけれど，便利な関数のいくつかは，単純な整数と文字の配列の初期化を認めるために与えられている．整数値へ連続する数のかたまりをセットするために，関数initwを使用する．文字列のためでなく，文字のためには(0－255の間の単一バイトの整数)，initbを使用しなさい．

たとえば，

```
int foobar [10]= {3, 2, -2, -5, 3, 6, 9, -23, -14, 0} ;
```

というUNIX Cの構文をシミュレートするために，まず，

```
int foobar [10] ;
```

を行うことによって,普通foobarを宣言できる．それから，命令文，

```
initw (foobar, "3, 0, -2, -5, 3, 6, 9, -23, -14, 0");
```

をmain関数の中にインサートする．

次の要項は，最適効率のために努めるとき，心にとめておくべきである．

1. コメントは，そのファイルがディスクから読まれるときに，動的にソースファイルから取り外されるので，プログラムを十分に文書化しないことは許されるものではない（それは怠惰である）．
2. switch文は，switch内の変数（たとえば" switch (xx) …" の中のxx)がcharとして宣言されるとき，最も効率的である．整数値は，しばしば，ファイルI/Oを含むアプリケーションを処理しているテキストの中に，文字値をかかえ込むのに使用される．大きなswitchの前に，そういった値を文字変数へ割り当てると，メモリーをセーブすることができ，実行をスピード・アップする．
3. switch文の中のcaseは，その出現する順にテストされる．だから最も共通とするケース（あるいは，最も速い応答時間を必要とするもの）は，先に現われるべきである．
4. 最高スピードでコンパイルを実行するために，CCコンパイラに－oと－e　xxxxオプションを与えるべきである．コードの長さを最も短くするためには，－e xxxxオプションだけをCCに指定すべきである．
5. Cの中の論理式は，値が実際に必要とされているときはいつでも，数値0（偽の場合），1（真の場合）へ評価する．しかし，処理の流れのテストで使用されるときは，どんな値にも全く評価しない．これが意味しているのは，ユーザーは多くの状況で，論理式の数値の結果を利用できるということである．次のコードの一部を考えてみなさい．その目的は，a<bの場合は1に，a>=b:の場合は0に，変数xをセットすることである．

```
if (a <b) x=1;
  else x=0;
```

同じ操作は

```
x= (a <b);
```

と書かれうる．

これは，サブ式“(a <b)”が自動的に必要な値に評価をする方法を利用し，2つの別々の割り当て式，それらの連想コントロール構造，すべての伴っている相当なオーバーヘッドといったものの使用をさける．

6．簡略化に関連する機会は，変数がゼロと同等か不等かをテストされるのが必要なときはいつでも生じる．どんな式でも，論理的には“真”とみなされるので，その式がゼロ以外の値に対して評価する場合，“a!=0”のような式の“!=0”の部分は,特に余分なものである．

```
if (a!=0)printf (" A is non-zero") ;
```

あるいは，if (a==0) printf (A is zero");
のような命令文は，

```
if (a) printf ( " A is non-zero);
if (!a) printf ( " A is zero");
```

と同様に書かれる．

もちろん，そのような簡略化は，与えられた状況にいつも適切とは限らない．問題の変数が，向かうカウンタとして使用され，多くの異なった値を引き受けることを期待されている場合，“a!=0”といった方が，そのプログラムの論理に対してよりクリアであるかもしれない．変数がブール・フラッグとして使用されたり，ゼロの値がある意味で特殊に考えられたりする場合，短目のフォームの方がクリアであるし，事実，より短目のオブジェクトコードへ導く場合もある．

付録　B

エラーメッセージの説明

B.1 CC エラー・メッセージ

この文書の中では，directory という項目は，任意の CP/M の論理ドライブとユーザー領域の組みわせを表わすのに使用される．

ファイル I/O エラー

close error　ディスク・ドライブ・ドアは開いているか？　もしそうでなければ，おかしなハードウェアの問題をかかえていることになる．

Error on file output…disk full?

もしもそうでなければ，ハードウェアをチェックしなさい．

Can't find CC2, COM;writing CCI file to disk

2つのディレクトリがあり，そこで CC が CC2.COM をさがし出そうとする．その2つのうちの1つは，常に現在のディレクトリであり，もう1つは－a オプションが CC で使用されるか否かにかかっている．もしそうならば，オプションで指定されたディレクトリが検索される．さもなければ，デフォルトのディレクトリ（第1章の設定セクションで定義されたように）が検索される．CC2.COM が検索される2つのディレクトリの中で見つからない場合，このメッセージがプリントされる．

DISK read error

新しいフロッピーをフォーマットするときでは?

Cannot open : 〈filename〉

指定されたファイルがみつからない．ユーザーがアングル　ブラケットで囲まれた# include ファイルを求めて，特殊なディレクトリを検索するためにCCを設定したのなら，ユーザー・ナンバー／ディスク指示子はこのエラー・メッセージの中でファイルネームを先導してプリントされる．CCが設定されなかったのなら，ディスク指示子だけが現われる．ユーザー・ナンバーの接頭辞は，CCのコマンド・ライン上では許されないので，トップ・レベルのソース・ファイルは，異なる論理ドライブ上にあるけれども，CCが呼び出されるときに現在のユーザー領域の中に常になければならない．

オーバーフローの状態

sorry;out of memory

ソース・ファイルが長すぎて，メモリーに入りきらない．メモリーをもっと大きくするか，もし可能ならそのソース・ファイルをもっと小さくくだくかする．

Out of symbol table space;specify more…

CCでシンボル・テーブル・スペースを予約するために，－r オプションを使用しなさい．あるいは，そのソースのファイルをもっと小さいものにくだきなさい．

Too many functions (63 max)

単一のBDS Cのソース・ファイルに63以上の関数の定義を含んでいる．63以上の関数を持っているプログラムを，別々のソース・ファイルへ分割されなければならない．

String too long (or missing quote)

たいていこのエラーは，文字列のまわりに二重引用符をつけるのを忘れているときに起こる．文字列が正しく区切られているようであれば，文字列の範囲内にエスケープされていない（バックスラッシュでそれを先行している）二重引用符記号を含んでしまっていないかどうかをチェックしなさい．

Too many cases (200max per switch)

Case 文が200をこえている．

include files nested too deep

これは，再起包含をしようとする場合に起こったりする．

String overflow;call BDS

これは，# define 命令の中でとても長い識別子をあまり多く持ちすぎたために起こる．プリプロセッサの文字列テーブルのオーバーフローである．これはとてつもなく大きなプログラムのときにのみ起こってしかるべきである．より大きな文字列のスペースの割り当てを持つコンパイラの特殊なバージョンは，ある種の BDS C パッケージ購入証明と共に，BD ソフトウェアへ SASD（self-addressed stamped disk，マスターディスケット）を送れば得られる．

プリプロセッサエラー

Warning:Ignoring unkhown preprocessor directive

サポートされていないプリプロセッサディレクティブに出くわすと，この警告がプリントされる．現在のところ，これが唯一の非致命的な診断メッセージである．

Eof found whem expecting # endif

条件付きコンパイルが不当に区切られている.

Not in a conditiondl block

if # ifdef がないのに, # endif がある.

Conditional expr bad or beyond implemented subset

CC は, # if プリプロセッサの中で, 演算子のサブセットしか使用できない. if 式の構文の要約に関しては, 第4章をみなさい.

Bad parameter list element

悪い識別子が, 関数定義のパラメタ・リストの中に現われた.

Missing parameter list

パラメタ化された# define からの識別子がそのパラメタなしで現われる.

Parameter mismatch

パラメタ化された# define からの識別子が, その定義の中のものとはちがったパラメタの数で使用されている.

Missing legal identifier

識別子が式の中で期待されているのに何も現われない.

構文エラー

注: 終結していないコメントは, コンパイラからあらゆるかわったエラーメッセージを引き出すことができる. 次のメッセージのうちの一つが現われて, その原因がわからないのなら, −p オプションを CC に与えてみなさい. そして, そのコードがプリント・アウトの中でのあるポイントで"カット・オフ"し, そうであれば, チェックしなさい. もしそうならば, それはたぶん, 閉じられていないコメントの位置である.

というのは，次のテキスト全部は，コメントの部分として解釈され，プリント・アウトの前にソース・ファイルから除外される．

Encountered EOF unexpectedly(check curly-brace balance)

閉じていないコメント，そして閉じていないカーリー・ブレイスをチェックしなさい．ユーザー・グループのプログラム LCHECK.C は，レベルを入れ子にするカーリー・ブレイスをチェックするのに使用される．

Unmatched right brance

左のブレイスがないか，あるいは関数のない右ブレイスがあるかのいずれかである．

Illegal external statement

これはたいてい，関数の中に右ブレイスが多すぎたために起こり，コンパイラに早まった関数定義の末尾の検知をさせる．

Function definition not external

これは，何か関数定義のようにみえるものに，別の起動可能な関数定義の範囲内で出くわしたときに起こる．たぶん，それは関数コールの後にセミコロンがないだけか，あるいは類似した型がある場合である．

Missing semicolon

このエラーが意味しているのは，たいていそれが何かを言うということである．しかし，セミコロンを忘れていると，あまり意味のないエラー・メッセージを引き起こす場合もある．

Expecting (

大体　while if switch というキーワードの後に出くわす．

Unmatched left parenthesis

これは，大抵検知されるエラーのうちの別なタイプであるが，ある場合他のあまり意味のないメッセージを生成するかもしれない．

I'm tolally confused Check your control structure!

これは，関係のない文字，あるいは誤ったカーリー・ブレイスのネスティングによって起こる．

Illegal { encountered externally

不適当なカーリー・ブレイスによって起こる．

Mismatched control structure

テーマを入れ子にしている不等なカーリー・ブレイスにおける別なバリエーション．

Expecting while

は，do…while 文にその while が欠けている．

Illegal break or continue

break 文は，Switch　文とループの内側で許されるだけである．continue 命令文は，ループの内側でのみ許される．

Bad for syntax

自明のことであるが，セミコロンの正しい数(2)とその配置をチェックしなさい．

Expecting { in switch statement

switch 文の式の部分は，カーリー・ブレイスの中の複合命令文によって読まれなければならない．

Bad case construct

各 case の定数は，絶対定数か単純定数式(文字定数はもちろん受け入れられ

る）のいずれかでなければならない．

Illegal statement

このエラーは，たとえばcaseやdefault文が，switchコンストラクトの外側にみつかるときに起こる．

Syntax error

このエラーを引き起すのは，大体不明瞭な何かである．たとえば，文字列の前に左の二重引用符がない場合，ファイルの中の関係のない文字もこれを引き起こす．

Bad constant

switch式やcase定数のために使用される値のように，定数式でなければならない式がいくつかある．

Bad octal digit

ゼロで始まる数字の8進定数が，8.9という数字を含んでいると，このエラーが起こる．

Bad decimal digit

10進定数が，悪い文字を含んでいたり，ユーザーが順序0xでもって，16進定数を先行するのを忘れていると，これが起こる．

Curly-braces mismatched somewhere in this function

これはむしろ，コンパイラにとって有用な特徴である．ソース・テキストが左のカーリー・ブレイスを多く持ちすぎていると，このエラーは最初に起こったミスマッチを検知した関数の始まりに対してさし示す．

宣言エラー

Undeclared identifier: <name>

これは単に宣言されなかっただけの本当の識別子であるかもしれない．あるいは，識別子のスペルのまちがいかもしれない．

Bad declaration syntax

たいていコンパイラは，型指定子 (cher，int のような) をみつけると，すぐにそれは，データ宣言を処理していると思う．このエラーは，そのキーワードを含む命令文の残りの部分が宣言と似てない場合に起こる．

Need explicit dimension size

配列が関数に対してフォーマル・パラメタであるときにのみ，配列宣言の中で次元サイズを省くのが許される．そういった配列が二次元なら，その第一次元だけが省かれる．

too many dimensions

BDS C　は，一つの配列変数につき，最高 2 までの次元を認めている．

Bad dimension value

配列宣言の中の次元は，変数としてあるいは定数式として与えられなければならない．

Redeclaration of : <name>

単一変数のための複数の不一致な宣言を，実際に書き込むことは別にして，このエラーを引き起こすのは，ボディーの直前にするかわりに関数のボディーの内側で関数のパラメタを宣言するとこうなる．その関数のボディーの前に宣言されなければ，パラメタは自動的にタイプ int を与えられることに注意しな

さい．ゆえに，関数のボディーの中の局所変数として，フォーマル・パラメタ識別子のその後の宣言は再宣言を構成する．

Expecting in struct or union def

構造体または共有体で｛がない．

Illegal structure or union id

構造体宣言の構造体タグの位置に現われる識別子が，先の構造体タグとちがう何かとして宣言されたときに生じる．

Attribute mismatch from previous declaration

大きな属性（タイプとオフセット）を与えている他の構造体の範囲内で，再使用される構造体宣言の中のエレメントは，各構造タイプの範囲内では同じである．構造エレメント・ネームが異なる属性を用いて再使用されるときに，このエラーが現われる．

Declaration too complex

あまりに多くのレベルの間接的手段があるときに，このエラーが起こる．あるいはコンパイラがあまりに多くのかっこを操作するときに起こる．

Missing from formal parameter list: <name>

関数ボディーの前にパラメタの宣言が現われると，これが起こる．しかし，そういったパラメタは，関数名につづくパラメタ・リストの中に現われない．

Bad parameter list syntax

関数定義のパラメタ・リストの中のコンマで区切られた識別子のリスト以外の何ものかがこのエラーを引き起こす．

いろいろなエラー

<text> :open error

CCがまちがって形づくられたコマンド行のオプションを検知する場合に，このメッセージと共に理解できなかったテキストをプリントする．正しいフォームを確認するために，第1章の中のコマンド・ライン・オプションの記述をチェックしなさい．

Compilation aborted by control-C

コンパイルの間にユーザーが，コンソール上でコントロール－Cをタイプする場合，このメッセージがプリントされ，コントロールはコマンド・レベルに戻される．コンソールポーリングは，第1章の設定セクションでのべられたように，CC.COMの特殊な設定によって割り込み不可能になることに注意しなさい．これは，割り込みを受け付けるシステムのためにコンパイラの実行中にタイプ・アヘッドを認める．

Can't have more than one default:

1つ以上のdefaultがあると，これがプリントされる：　句は特別なswitchコンストラクトのために出くわす．

Illegal colon

コロン（文字ストリングの中以外に）は，3進演算子の部分として認められているだけである．あるいは，case, default といったラベルの後につづく．

Undefined label used

ラベル・リファアレンス（goto文の中でのみ認められている）は，現在の関数定義に対して局所的なラベルを参照する．

Duplicate label

特別な識別子は，一つの関数ごとに一つのラベルのために使用されるだけである．

B.2 CC2 エラー・メッセージ

CC2 のプリントしたファイル I/O エラーのいくつかは同じであるか，または CC のためにあげられたメッセージと非常に類似している．だから，このセクションでは繰り返して述べない．

ファイル I/O　構文，オーバーフロー，その他のいろいろなエラー

Can't create CRL　file

出力ドライブ上のディレクトリがいっぱいになった？

CRL Dir overflow:break up source file

各 CRL ファイルに割り当てられたディレクトリ・スペースは，512バイトあるだけである．ネームの中に8.あるいはそれ以上の文字を含んでいるように定義された関数（各ネームの最初の8つの文字だけが実際にディレクトリの中に記憶される）をあまり多く持ちすぎると，単一のソース・ファイルのためにディレクトリ・スペースをオーバー・フローすることがある．関数名を短かくするか，一つのソース・ファイルごとの関数の数を減らしなさい．

Internal error: garbage in file or bug in C

これが CC2 の間に起こったなら，それはたぶんコンパイラ・バグである．援助の必要なときは，BD ソフトウェアに連絡して下さい．

Illegal statement

大体，ワイヤされた何かに出会った．

Missing { in function def

このエラーを引き起こすものは，どれも本当に関数定義のスタートではない．しかし，わけがあって前の（あるいは現在の）関数が終結してしまったと，コンパイラが考えて，別のものが始まる．プログラムの中に多く右カーリー・ブレイスがありすぎないか，どうかをチェックしなさい．

Missing semicolon

式，命令文の後にセミコロンがないと，たいていは検知され正しく診断される．

Sorry, out of memory, Break it up!

ファイルが長すぎる．たいていの場合 CC を通るものは，CC2 も通るが，例外もある．

The function 〈foo〉 is too complex;break it up a bit

あまりに大きな関数を操作することのできないある内部のテーブルがある．いろんなテーブルやリストのために，どれほどのスペースを予約するのかを，コンパイラに教えている．ひとかたまりの混乱しているパラメタをユーザーがセットするように要求するよりも，むしろほとんどのテーブル・サイズ定数をセットし公正に重い関数を認めるように決めた．しかし，ある程度までである．正しく構成されたCプログラムは，このメッセージを出すべきでない．

Sub-expression too deeply nested

このエラーのほとんどの原因は，永遠につづく多重割り当ての命令文である．解決法は単にそのラインを小さめのものに分けるとよい．

Compilation aborted by control-C

CC の適切な設定バイトが，ユーザーによってゼロにカスタマイズされなけ

れば(チャプター1の設定のセクションをみなさい)，システムコンソールでコンロール－Cをタイプすると，コンパイル作業は終結し，このメッセージをプリントし，すぐにコマンド・レベルに戻る．

式におけるエラー

Lvalue required

オブジェクトは，アドレスを用いるか，割り当て演算子の左で正しくなければならないことを要求される．

Lvalue needed with ＋＋ or －－ operator

単純変数だけが，自動増加，あるいは自動減少されうる．

Bad left operand in assignment expression

割り当て演算の左にある式が，それに割り当てられた値を持つことができないと，このエラーが起こる．たとえば，文字列よりはLvalueではないが，正しいLvalueを生み出すために添え書きされる．

Mismatched perenthesis

左のペアレンセスにつづく式は，マッチしている右のスクエア・ブラケットの後すぐに続けられない．

Bad expression

これは，式（あるいは式になると想定されるもの）が，コンパイラにとって意味をなさないときにプリントされる一般的なエラーメッセージである．これは必ずしもエラーが明白であることを意味しない．しかし，たいていはそうである．

Bad function name

これは，コンパイラが左のペアレンセスの直前の識別子を見たときに，その識別子が先に関数名以外の何かとして宣言されてしまっているときにプリントされる．

Bad arg to unary operator

単項演算子のオペランドは，その演算子にとって適切なタイプのうちに入らない．

Expecting :

？：式を書こうとしたのか，またはコロンを含むのを忘れたかである．

Bad subscript

ポインタにとって正しいタイプの配列添え字は，算術演算か？　たとえば，配列式の中での添え字はポインタであり得ない．

Bad array base

添え字のできないことを添え書きしようとしていることになる．一般的な原因：main 関数の中の argv を正しく宣言していないのに，添え書きしようとしていないか．

Bad structure or union specification

．(ピリオド)演算子の左に対する式は正しい構造体，あるいは共有体のベースではない．

Bad type in binary operation

あるタイプの変数は，二進の演算子の中ではいっしょに現われない．たとえば，2つのポインタを加えることはできない（それらを減算すると，目的の値の始まりがさし示すサイズによって，概算された結果を生む)．あるいは，単一変数以外の目的の値では，ほとんどのビット・ワイズのそして不明瞭な演算を

する。

Bad structure or union member

.（ピリオド），または→の演算子の右に対する式は，有効な構造体でも共有体エレメントでもない。

Bad use of member name

構造体や共有体のメンバーとして宣言された識別子は，構造体や共有体の操作の外側で使用されえない。

Illegal indirection

目的はポインタではないのに，それがポインタであるかのようにいくつかの目的で操作するように試みがなされている。

Encountered EOF unexpectedly

これは，悪い構文のエラーか，ファイル損失のサインのいずれかである。コンパイラの現在のバージョンは，たいていこの種のエラーに関してもっと具体的であるけれども，うまくマッチしていないカーリー・ブレイスも原因になるかもしれない。

Bad argument List

関数コールのためのパラメタ・リストの中で，不当なものがみつけられた。たとえば，式の中で正しくないセミコロン，あるいは他のキーワードなど。

Missing misplaced (

while キーワードなどの後の ( ) 内の式がない。また，左かっこがみつからなかった。

Missing or misplaced)

左のペアレンセスで始まる式は，閉じの右かっこによってその後を続けられなかった．これは，式のまん中での関係のない文字のせいかもしれない．

B.3 CLINK エラーメッセージ

注：　CLINK によってプリントすると思われるファイル I/O のエラーの多くは，説明の必要はないが，注釈の必要なものだけをここにあげておく．

No user area prefix allowed on main filename

ユーザー領域の接頭語は，CLINK コマンド行における第1番目のものを除いて，すべてのファイルネーム上で認められない．

Dir full

新しい出力ファイルをつくるためのディレクトリ・スペースがない．

Error writing: <filename>

たぶん，ディスク上のデータ・スペースがなくなった．

Can't close: <filename>

ハードウェアのエラーか？

No main funetion in <filename>

CLINK コマンドライン上で指名された最初の CRL ファイルは，ユーザーがリンクしようとしているプログラムのために，main 関数を含んでなければならない．L2 リンカー（ユーザーズ・グループから手に入る）には，この規制がないことに注意しなさい．

Missing function (s) : <関数名>

表示された関数は，コマンド行でリストされたファイルのうち，あるいは標準ライブラリ・ファイルの中でみつからなかった．ファイルをロードするのではなくて，走査するために－f オプションを使用した場合，指名された関数のいくつかは表示されうるが，ロードされない．前の関数が，それらを参照しなかったからである．この場合，落としていた関数を含んでいたファイルを再走査するだけでよい．

Warning! Externals extend into the BDOS!

外部データ領域の終りアドレスが，BDOS のベースより大きいときに，これがプリントされる．そのコードが，別の環境で実行される予定なら，このメッセージは無視される．しかし，リンカーがこのメッセージをひき起こしたようなシステムで，そのプログラムを実行しようとしてはならない．

Warning! Externals overlap code!

外部データ領域の開始アドレスが，プログラムの最後のコード・アドレスより小さいか，あるいは等しいとき，これがプリントされる．たいていこれが意味するのは，外部領域が－e オプションでもって，あまりに低く置かれたということである．そのコードが外部領域の上に常駐するカスタマイズされた環境のためのものなら，そのメッセージを無視しなさい．

Out of memory

リンクするにはメモリーが不十分である．L2 リンカーを使用してみなさい．それは，CLINK よりも最高 8 K大きくプログラムをリンクできる．

Bad symbols

－y オプションの使用によって，読み込まれたシンボルファイルがうまく書式化されていないエントリーを含んでいる．

Ref table overflow

前方参照テーブルのスペースがなくなった．もっとスペースを予約するために，－rオプションを使用しなさい．使い方は“-r xxxx”である.ここでxxxxは16進で与えられる．600はデフォルトのものである．エラーがなくなるまで，800やA00などを試してみなさい．

SYM file symbol already defined: <symbol>

－yオプションの使用によって読み込まれるシンボルは，すでにロードされ定義された関数と同じである．元の値は，キープされる．というのは，その関数はすでにロードされているし，定義されている．もしくはその一方であるからで新しいものは捨てられる．

Ignoring duplicate functon: <name>

ロードされているCRLファイルの中の関数は，前のファイルからすでにロードされた関数と同じ名前を持っている．オリジナルはオープンされ，新しいバージョンは無視される．

Sorry : 255 funcs maximum

CLINKは，単一のリンクにおいて最高255までの関数を操作できるだけである．もしもっと多くの数の関数をリンクする必要があるなら，BDS Cユーザーズ・グループからL2リンカーを購入しなさい．

付録　C

初心者のCプログラマーが起こしやすいまちがい

C言語には，いろんな側面があり，はじめて取り組むものは多くの問題をひき起こしがちである．このセクションで私は，困惑したユーザーが電話や手紙でひっきりなしに持ち込む複雑な問題，Cの“特徴”についてまとめてみようと思う．

C.1　“＝”と　“＝＝”

演算子＝は，割り当てのためにだけ使用され，演算子＝＝は，“等しい”に関係のある状態をテストするのに使用される．２つの演算子は，それらを表現するために使用される文字を除いて，共通するものは何も持っていないが，混乱したときにデバック時間を無駄にすることもある．

Cで共通しているコンストラクトは，より大きな式の中で記憶された割り当て操作をすることであり，たぶん条件語句を含んでいる．これは，

```
if ((c=getchar ( ))== '\n')
    print f (" You typed a newline!\n");
```

のような命令文へと導くことができる．ここで，Cの初心者は＝の操作をそれが実際にある割り当て式のかわりに，条件付きのテキストとして解釈するかもしれない．

次のコード分割を考えてみなさい.

```
if (!(c=getnext( ))) {
 printf(" All done \n ");
     break;
}
```

この命令文の中の if 式は, getnext 関数から変数 c へリターン値を割り当て, そのリターン値がゼロかどうかを調べる. もしゼロなら, " All done " をプリントし, break する. もちろん, 疲れたプログラマーがこれを非常にはやくみたなら, それは c が getnext のリターン値と比較されているかのように思われるかもしれない.

C.2 配列の添え書き

Cでは, 長さ n の配列は, 0 から n − 1 までの数字のついたエレメントを持っている. ユーザーが長さ n の配列を宣言し, 値 n の添え字でエレメントを参照しようとするなら, ユーザーはその配列の終りを過ぎてデータを参照していることになる. これが一番ひんぱんに起こるのは, ユーザーが BASIC 言語の面からものを考えているときである. BASIC では, 長さ x の配列はエレメント・ナンバー 0 とエレメント・ナンバー X の両方を持つ (最近の BASIC では option base 文によって低数を変えられるが…). C において一番一般的な for-ループのコントラクトは次の

```
for (i=0; i <n; i++)
     …
```

のように 0 から n − 1 までの数字のついた n 項目を通してくり返す. そして, そういったループは, 配列を通してくり返すには理想的である. もし, n 項目のために, 本当に 1 から n まで数字のついた配列を持つ必要があるなら, 要求

されているのよりも一つ多い項目を持つように配列を宣言し，未使用の0番目のエレメントを残さなければならない．

C.3　いかにしてポインタを使用せずにおくか

ポインタ変数が外部か，関数内のいずれかの，プログラムの中で宣言されるとき，それは自動的に値を与えられない．ポインタは16ビットの変数であって，データの他の部分のアドレスを保つために使用される（そしてさし示すため）．そして，変数と同じく，使用の前に初期化されなければならない．一番よくみうけられるミスは，初期化されていないポインタを通して，間接的に値を割り当てることである．たとえば，宣言

```
char * foo;
```

は，foo が初期化される前に

```
* foo= 'a';
```

のような命令文が後につづいたとき，予測しなかったことが起こり始める．上記の割り当て命令文のいっていることは，“変数 foo の値によって指定されているアドレスを持つロケーションに文字'a'を置きなさい”ということである．foo が何ものに対しても初期化されていなかったら，文字'a'は，メモリーの中のランダムなロケーションに記憶される．ここで，正しい手順をいうと，バッファ領域を宣言し，その領域のアドレスに foo を割り当て，foo を通して間接的にデータ割り当てを始める．たとえば，次の順で文字'a'を buffer [0] のロケーションに置く．

```
char buffer[50] ,* foo;
foo=&buffer;
......
* foo= 'a';
```

C.4 関数はポインタをその自動データに戻すべきでない

関数が呼ばれたプログラムに戻り次第，その関数に対して局所的だった記憶（たとえば，すべての宣言された局所変数が記憶された場所）は，割り当てを解除され，次に呼ばれる関数による使用のために準備される．一般的なミスはある関数（それを foo と呼ぶ）に，局所的なバッファの中である文字列を作らせ，その文字列のポインタを戻させることである．foo から戻ってすぐに，その文字列はもとのままである．しかし，もっと後のプログラムの過程では（その文字列が常駐しているスペースは，他の関数の局所データエリアのために割り当てられるように），その文字列は不要部分にかわる．こういった問題の解決法は2つある．

a）foo に，ストリングの結果をどこにおくかを教えるパラメタを取らせなさい（この場合，呼び出すものは foo のためにバッファを備える）．あるいは，

b）外部領域に文字列の格納場所をつくる．

各方法は，その利点がある．各コールで自分の格納領域をパスすると，戻された文字列が異なる領域のメモリーで別々にセーブされるようになる．一方，外部格納領域は，1つ少ないパラメタがパスされるように要求することによってコールする順を短かくする．しかし，何を行なおうと，その関数が戻された後でも有効のままでいるために，コールされた関数によって，局所的に割り当てられたデータを期待してはいけない!!

C.5 フォーマル・パラメタのきまり

そもそも"フォーマル・パラメタ"とは何か？　フォーマル・パラメタは関数がコールされるときは，いつでもパスすると思われている引数の1つである．すべてのフォーマル・パラメタは関数名のすぐ後に続く（ ）で囲まれたリストの中で指定される．関数のボディーの始まりを表わしている一つめの{の前に，そして（ ）で囲まれたリストのすぐ後に関数のフォーマル・パラメタの宣言がなされなければならない．

明白に宣言されていないパラメタは，単に int の値とみなされる．フォーマ

ル・パラメタが偶然，実際の関数のボディー（{ } の内側）で宣言される場合，コンパイラは，“再宣言”エラーと診断するだろう．というのは，フォーマル・パラメタのための宣言とみてもいないのに，フォーマル・パラメタの宣言がパスされ，コンパイルがその関数ボディーを処理し始めると，そのフォーマル・パラメタはintとして自動的に宣言されてしまうからである．

関数の呼び出しがおこるときはいつでも，フォーマル・パラメタの値のコピーがその関数へパスされる．そういった値のすべては，BDS Cバージョン1に関する場合，長さ16ビットである．これが意味しているのは，本来16ビットの大きさをもたない構造体，配列，データタイプは関数へ直接パスされえない．そういったデータ・タイプに対してポインタであれば可能である．さて，配列名が関数へパスされるとき，何が起こるであろうか？　配列へポインタをパスするにあたって，混乱するかもしれないが，特別なそして不思議なメカニズムがある．それは，ポインタが実際にパスされている宣言構文からいって直観的に明白でない．たとえば，次の関数を考えてみなさい．

```
int arraysum (array)
int array [3] ;
{
        retern array [0]+array [1]+array [2]
}
```

arraysumが，3エレメントの配列をフォーマル・パラメタとしてとっているようにみえるかもしれないが，実際はその配列に対するポインタがパスされる．その宣言は，全配列がパスされているかのようにみえる．しかし，配列の中でエレメントをチェンジする場合，ここでは，コールしているプログラムから受け取ったエレメントをチェンジしている．配列は一つだけ存在している．

フォーマル配列パラメタに関して，もう一つひっかかりやすいポイントは，実際に配列名をコールされた関数の中での単一ポインタ変数として取り扱うことができる，ということである（たとえば，別の配列のアドレスをそれに割り

当てると，それは他の配列のベースになる）．しかし，そういったものは，その配列が実際の（非フォーマル・パラメタの）配列であるときには作動しない（事実，予想しない結果になる）．カーニンハン＆リッチーの本は，ポインタと配列の“双対関係”に関する完全なチャプターをし含んでいる．このメカニズムはＣの最もパワフルで，最も混乱しやすいものである．

C.6　関数コールは（ ）を持つ

関数名は，引数のリストなしで使用される場合，出てきた式は指名された関数のアドレスとして評価される．そのリストが空であっても，その名前が（ ）で囲まれたパラメタのリストを後につづけられなければ，その関数に対するコールはなされない．たとえば，次の式，

```
i=endext( );
```

は，外部データ領域の終りのアドレスを変数 i:に割り当てる．

一方，次の式

```
関数    i=endext;
```

は，endext のアドレスを変数 i に割り当てる．だが，endext が先に宣言されている場合だけである．

endext が先に宣言されていなければ，コンパイラは endext を“宣言されていない変数”として診断する．一つめの方の式では，endext は int を戻す関数として暗黙のうちに（文脈の中で）宣言される．

付録　D

Cプログラムにおけるダイナミックオーバーレイ

execあるいはexecl（は，事実動作するが，真の区分化ツールよりももっと“チェイン”操作に似ている）に頼らずに，Cプログラムが物理メモリーよりも長くいられるようにするために，CLINKプログラムはプログラム区分化を可能にするためにくふうされている．一般的な考えは，Cのランタイム・パッケージ，“main”関数，あるいは一つ以上のオーバーレイ・セグメントが必要とするかもしれない他の関数を含むメモリー（TPAのベースで）で，ルート・セグメントの一つのコピーを常にとどまらせることである．そのルート・セグメントはより高いメモリーの中でのオーバーレイ・セグメントのロードをコントロールし，ルート・セグメントの上のどこかにあるメモリーへロードするとき，各オーバーレイ・セグメントは，低レベル・オーバーレイ・セグメントの中の関数のエントリー・ポイントだけでなく，ルート・セグメントの範囲中でのランタイム・パッケージのエントリー・ポイントも利用することができる（ルート・セグメントも）．

たいてい（たとえば，オーバーレイが使用されないときなど），実行しているCプログラムの実行時の環境はこのようにみえる．

low memory:　　base＋100h:　C.CCC run-time utility package (csiz bytes)

ram＋csiz:　start of program code

```
                                    ...(program code)...
                        xxxx-1:  end of program code

                          xxxx:  external variable area (y bytes long)
                                 ... (external data)...

                        xxxx+y:  free memory,
                                          available for
                                                   storage
                                                            allocation

                          ????:  as low as the machine stack ever gets
                                          local data,function parameters,
                machine stack:            intermediate expression results,

                                          etc.
high memory:              bdos:  machine stack top (grows down)
```

Memory Map 1.

xxxxは，プログラム・コードにつづく最初のロケーションであり，yは，外部変数に必要なメモリー量である.

オーバーレイを組み込ませるために，まずオーバーレイ・モジュールをどこに常駐させるかを決定しなければならない. 初期のBDS Cのバージョンは，外部領域の終りから発生する局部データ・フレームを持っていた. これは，オーバーレイ・モジュールを安全にロードするためのメモリー・ロケーションの決定をむずかしくした. オーバーレイを操作するために，提案されたことは一番大きなオーバーレイ・モジュールの組み合わせを適応させるために，ルート・セグメント・コードの最後と，外部データ領域のスタートとの間に十分な余地を残すということである.

現在，BDS Cは，メモリー・スタック上に局所データの領域を割り当てる．次にあげている修正されたメモリー・マップは，この方法のオーバーレイ操作を適応させている．

```
low memory: base+100h: C.CCC run-time utility package (csiz bytes)
                 ram+csiz: start of root segment code
                           …(root segment code)…
                   zzzz-1: end of root segment code

                     zzzz: start of overlay area
                           …(overlay area)…
                   xxxx-1: end of overlay area

                     xxxx: external variable area (y bytes long)
                           …(external data)…

                   xxxx+y: free memory,
                                 available for
                                       storage
                                             allocation

                     ????: as low as the machine stack ever gets
                                 local data, function parameters,
           machine stack:        intermediate expression results,
                                 etc.etc.
high memory:         bdos: machine stack top (grows down)
```

Memory Map2.

zzzzは，オーバーレイ・セグメントがロードされる場所で，最長のセグメン

トは xxxx にとどかないと保証されている。

バージョン1.5において，外部データ領域の後にオーバーレイ領域をおくことも可能である（しかし安全でない）。この代替的構成のためのメモリー・マップは次のようになる。

```
low memory:     base+100h: C. CCC run-time utility package (csiz bytes)
                 ram+csiz:   start of root segment code
                            …(root segment code)…
                   xxxx-1:  end of root segment code

                     xxxx:  external variable area(y bytes long)
                            …(external data)…
                 xxxx+y-1:  end of external data area

                   xxxx+y:  start of overlay area (ssss bytes long)
                            …(overlay area)…
           xxxx+y+ssss-1:   end of overlay area

             xxxx+y+ssss:   <unused memory>

                     ????:  as low as the machine stack ever gets
                                    local data, function parameters,
            machine stack:   intermediate expression results,
                                    etc.etc.
high memory:         bdos:  machine stack top (grows down)
```

Memory Map3.

記憶割り当て（alloc と sbrk）は，いつも外部データの末尾の後にすぐに続く領域からメモリーを獲得し始める。ユーザーのプログラムの中で，記憶割り当

て関数（alloc, free, sbrk, rsvstk）を使用するつもりなら，オーバーレイ領域のサイズである引数ssssを持つsbrk関数をまずコールすることを忘れないように，でなければ，記憶割り付けルーチンはオーバーレイ領域の範囲内からメモリーを割り当て始める．

この文書の残りがわき道にそれたりしないようにするために，私は元のオーバーレイ案は，メモリー・マップ2で示されたように実施されているものとする．

BDS Cに関する"ルート"セグメントと，"オーバーレイ"セグメントのつくり方についてちょっと延べてみよう．まずはじめに，ルート・セグメントで定義されたすべての関数は，オーバーレイ・セグメントによってアクセスしやすいようにしたい．これは，ルート・セグメントがリンクされるときにすべての関数アドレスを含んでいるシンボル・テーブル・ファイルをディスクに対しCLINKに書き出させることによって達成される．CLINKに対する－wオプションがうまくやる．このシンボルテーブルはスワップできるセグメントをリンクするとき，後になって使用される．

ルート・セグメントをリンクするとき，外部データ領域のロケーションをセットするのに－eオプションを使用しなさい．実行時で最大のオーバーレイ・セグメント・コードの末尾のすぐ後から[21]，外部データがスタートするとみなし，オーバーレイ領域と矛盾する（このように－eはメモリー・マップ3で示されたように2番目のオーバーレイ案を使用しているときにのみ省かれる）．

ルート・セグメントのコードの範囲内でスワップできるセグメントは

```
swapin (name, addr);   /* read in a segment…don't run it   */
```

ということによって，ディスクからメモリーへロードされる．ここで，addrはルート・セグメント・コードの最後のバイトにつづくロケーションである．ユ

21) 私は，低メモリーは"below"（下の）ハイメモリーであるという意味で"below"という単語を使用している．前出のメモリー・マップにおいて，belowはページのトップという意味である．

ーザーは－eオプションを与えたり，リンク後に－sによって出力されるサマリを読んだりせずに，ルートを一回リンクすれば，この値をみつけることができる．セグメントを実際に実行するためには，関数のポインタを使用して間接的にそれをコールしなければならない．

一つ例がある．我々はptrfnと呼ばれる関数のポインタを宣言し，ロケーション3000hでovllと名付けられたセグメントの中へロードする．そして，そのセグメントをコールするのである．その手順は次のようになる．

```
int (* ptrfn)( ):   /* can be whatever type you like */
ptrfn= 0 ×3000 ;
......
if (swapin ( " ovll " , 0×3000)!=ERROR) /*ロード・エラーのチェック*/
( * ptrfn) (args…);          /*エラーがなければ，それをコールする*/
  ......
```

オーバーレイ・コードは，コールされた後に値を戻さないかもしれないが，関数のポインタはある種の型でもって宣言されなければならないことに注意しなさい．何も思い浮かばない場合，intを使用しなさい．上記のように，セグメントが呼び出されるとき，コントロールはセグメントの"main"関数にパスする．パラメタが"argc"と"argv"といったフォームのものでなければならないということは全くない．"main"関数に関しては，最初にコールされたときに持っている性能以外，何も特別なことはない．ロードされたセグメントの範囲内の"main"関数は，そのセグメントにとって許された唯一のエントリ・ポイントである．

Swapin関数は，STDLIB2.Cに含まれている．外部データ領域上で，試みられたロードを検知するには，ロケーションram＋115hの内容を用いて，ロードされた最後のアドレスを比較すればよい．

もし低レベルで操作したことがないなら，整数（あるいは，符号なし）のポインタを用いた間接的方法を使用することによって，ロケーションBASE＋

115hの16ビットのアドレスの値を得ることになる．ロケーションBASE+115hは，外部データ領域の開始ロケーションに対するポインタを含んでいる．

さて，我々は実際にオーバーレイ・セグメントをつくるということを除けば，すべての操作方法もわかった．オーバーレイ・セグメントは基本的には，単なる普通のCプログラムであり，C.CCCランタイム・ユーティリティ-パッケージがオーバーレイ.・セグメントの前に付加されないことを除けば，ルート・セグメントと同じような"main"関数を持っている（ルート・セグメントにおけるC.CCCランタイムパッケージは共用される)．オーバーレイ・セグメントとルート・セグメントの他のちがいは，ロード・アドレスである．ルート・セグメントは常にTPAのベースからロードされるのに対し，オーバーレイ・セグメントはどこででもロードするようにつくられている．いったん，オーバーレイ・セグメントをロードしてしまったら，それをリンクするのにCLINKコマンドの特別なフォームを与える．

A> clink segment-name -v -l xxxx -y symbol-file [-s…] <cr>

ここで，segment-nameは，セグメントを含んでいるCRLファイルの名前であり，－vはオーバーレイ・セグメントを作ることをCLINKに対して示している（C.CCCが結合されないように)．そして，－l xxxx（16進のアドレスによって続けられる文字エル）は，セグメントのためのロード・アドレスを示す．－yオプションは，ルート・セグメントによってつくられたシンボルファイルを取り込む．これが省かれると，関数の新しいコピー"PRINTF"や"FOPEN"等がすでにルート・セグメントへリンクされてしまっていたとしても，CLINKはそれらを取り込む．ルート・セグメントからのシンボルテーブルを読み込むことによって，ルートの中ですでにリンクされたルーチンは，オーバーレイ・セグメントに対して有効にされるのは確実である．けれども，ルート・セグメントはシンボルテーブルの使用を通して，オーバーレイ・セグメントに属する関数について知ることはできない．それは，このパッケージの有効範囲を越えて互いに参照できるある種のリンクシステムが必要となる．

オーバーレイ・セグメントをリンクするとき，コンソール上にサマリを表示

するために－sを指定し，ルート・セグメントのシンボルファイルから読まれた文字だけでなくスワップできるセグメント自身のシンボルも書き出すために，－wを指定するかもしれない．この新しいシンボルファイルは別のレベルのスワッピングで使用され，そのように求められるべきである．

例をあげると：　ユーザーがプログラムROOT.Cを得たとする．これはスワップ・インしSEG1.Cを実行し，SEG1.CでもってSEG2.Cをオーバーレイする．ROOT.COMは100hからロードされ，3000hで終わる．我々は，3000hからそのセグメントをロードし，そして外部データ領域のベースを5000hへセットする（これは，どちらのセグメントも2000hより短いとみなしている）．

ROOTのリンクは

A> clink root －e 5000 －w －s <cr>

である．ROOT.COMがルート・セグメントであること（－vオプションは与えられなかったので），外部領域は5000hから始まり，ROOT.SYMと呼ばれるシンボルファイルを書き出すこと，サマリは，コンソールへプリントされることをCLINKに指定している．

各オーバーレイ・セグメントのリンクは，次のようになる．

A> clink seg1 －v －l 3000 －y root －s －o seg1 <cr>

これはCLINKにSEG1.COMがロケーション3000hからロードされるオーバーレイ・セグメントになること（－vがあるから），ROOT.SYMと呼ばれるシンボルファイルは前定義の関数のアドレスのために走査されるべきであること．リンクの後にサマリをプリントすること．オブジェクトファイルはSEG1として書き出されること（SEG1.COMにしないのはCP/Mコマンドとして，偶然それを呼び出すことをさけるため）といったことを指定している．

付録　E

CASM-アセンブリ言語-CRL ファイル変換プリプロセッサ

従来の BDS C では，アセンブリ言語のプログラムから再配置可能なオブジェクト・モジュール（CRL ファイル）をつくるための唯一の手段は，非常に複雑なマクロ・アッセンブラー（MAC. COM）を用いて使用しなければならないので，MAC を所有していないユーザーは全 BDS C パッケージとほぼ同じくらいの費用でそれを購入しなければならなかった．この文書は，"MAC"の必要性を取り除くために与えられている解説である．CASM はプリプロセッサで，非常にむずかしい CMAC. LIB よりはアセンブリ言語にもっと近いフォーマットで記述できる．".CSM"のアセンブリ言語のソース・ファイルを入力として取る．そして，標準のいたるところにある CP/M アセンブラ(ASM. COM)によってアセンブルされる"ASM"ファイルを書き出す．CASM は，自動的にどのアセンブリ言語命令が再配置パラメタを必要としているのかを認識し，適切な擬似操作や余分のオペコードを出てきた".ASM"ファイルへインサートするので，それは正しく CRL フォーマットへ直接アセンブルする．加えて，基本的な論理チェックが行なわれる．二重定義や未定義のラベルは検知されレポートされる．そして異なる関数の中の類似したラベルは認められ，ASM が異議を申し立てないようにするめに独特な名前へと転換される．

E.1　CASM. COM の生成

CASM は，BDS C パッケージにソース・ファイルとして含まれている．実行

可能のバージョンをつくるのにCASM.Cをコンパイルする前に，ユーザーのシステム構成に適合させるためデフォルトのライブラリ・ドライブとユーザー領域の定義，もしくはその一方をセットすることによって，ファイルをカスタマイズしなさい．CASMのコンパイルやリンクのための命令はファイルの最初の部分のコメントの中に含まれている．

E.2　コマンド・ライン・オプション

－cは，入力と出力の両方でコメントの保持を可能にする．デフォルトでは入力ファイルからすべてのコメントを取り除く．そして最後のASMファイルを形成するときにつけ加えられたアセンブリ・コードへコメントを入れない．－cが指定された場合，オリジナルのコメントを.ASMコードの新しいセクションへ付け加える．

－fユーザーが古いアセンブリー言語のソース・ファイルをCSMフォーマットへ転換するために古いCMAC.LIBマクロ・ライブラリー演算子に印をつける．

－o nameは，出力ファイルをname.ASMに指定する．普通出力ファイルは，.ASM拡張名をCSM入力ファイルのファイルネームに付加される．

CASMパッケージを構成するファイルは次のものである．

CASM.C　CASMプログラムのソース・ファイル

CASM.SUB　CSMファイルをCRLフォーマットへ変換するためのサブミット・ファイル

ASM.COM（またはMAC.COM) CASMの出力をアセンブルするための標準CP/Mユーティリティー

DDT.COM（またはSID.COM)　アセンブラのHEX出力を二進のCRLフォーマットに変換するための標準CP/Mユーティリティー

CASMが，.CSMファイルの中の特別なコントロール・コマンドとして認識している擬似命令は，次のものがある．

FUNCTION 〈name〉　　各関数はFUNCTION擬似命令で始まらなければならない．ここで，〈name〉は，.CRLファイル・ディレクトリの中の関数のために使用される名前である．他の情報はこのライン上にはあらわれるべきでない．CRLファイルの生成の古いCMAC. LIBの方法ではよくあったことだが，.CSMファイルのスタートで含まれている完全なリストを指定するようなことは，必要でないということに注意しなさい．

EXTERNAL〈list〉　　関数が他のCかあるいは，アセンブリーコード化された関数をコールする場合，これらの他の関数を指定しているEXTERNAL擬似命令はFUNCTION擬似命令のすぐ後につづかなければならない．一つ以上の名前がリストの中に現われても，そのリストは必要なだけのEXTERNALに展開される．関数命令だけがEXTERNALの行の中で現われる．データ名（Cプログラムの中で定義された"外部"変数のように）は，EXTERNAL"外部"命令文の中に配置することができない．

ENDFUNC

ENDFUNCTION　　この命令（両方のフォームは同じ）は，特別な関数のためのコードの末尾の後に現われなければならない．その関数の名前は，オペランドとして与えられる必要はない．いま，あげられている3つの擬似操作は".CSM"ファイルのアセンブリ言語命令の中で現われる必要のある単なる擬似操作であり，CMAC. LIBではよくあることだがアセンブリ命令自身が再配置のために変更される必要性はない．

INCLUDE 〈filename〉

INCLUDE " filename "　　この操作は，指定されたファイルが出力ファイルの現在の行からインサートされるようにする．そのファイルネームがアングル・ブラケット（たとえば，〈filename〉のように）で囲まれている場合，デフォルトのCP/M論理ドライブにその指名されたファイルを含んでいると仮定される(ユーザーのシステムにとって，具体的なデフォルト値はCASM. Cの中の適切な# defineを変更することによってカスタマイズされ)．そのネームがクォートで囲まれている場合，現在のドライブが検索される．ユーザーはたいてい.CSMファイルのスタートでファイルBDS. LIBを含んでいることが望まれ

るということに注意しなさい．だからランタイム・パッケージでのルーチンの名前はCASMによって認識され，未定義の局所的な前方参照として解釈されない．というのは，CASMは1パス・プリプロセッサで，オペランドとしてランタイム・パッケージのルーチン名を持つ命令のために，不必要な再配置パラメタを生成させることになるのである．擬似命令MACLIBはINCLUDEと同じなので，かわりに使用される．

.CSMファイルのフォーマットは次のようになる．

```
        INCLUDE             bds.lib

        FUNCTION            function1
  [     EXTERNAL            needed_func1[,needed_func2] [,...] ]
        code for function1
        ENDFUNC

        FUNCTION            function2
  [     EXTERNAL            needed_func1[,needed_func2] [,...] ]
        code for function2
        ENDFUNC
        .
        .
        .
```

追加の注意とバグ

1. ラベルは第1カラムから始まらなければならない．ラベルが第1カラムから始まらなければ，CASMはそれをラベルとして認識しないし，再配置は正しく行なわれない．
2. 再配置可能シンボルに対する前方参照は可能であるが，実行可能の命令

において等しいとみなされたシンボルに対するフォワード・リファレンスは認められない．このわけは，1パス・プリプロセッサであり，いつでも前もってわかっていないシンボルが命令の中で出会うためであり，CASMはシンボルが再配置可能でその命令のために再配置パラメタを生成するとみなす．

3. INCLUDE（とMACLIB）は，1レベルの包含にだけ作動する（入れ子にできない）．
4. 再配置可能の値がdwオペコードの中で指定される必要があるとき，それは特別なdw命令文の中で与えられた値だけでなければならない．さもなければ再配置は正しく操作されない．たとえば，16ビットの再配置可能の項目は，一つのdw命令文につき一つだけ認められる．
5. シンボル名で使用される文字は英数字に限られる．ドル・サイン（$）も認められているが，CASMの生成したラベルと矛盾したりするかもしれない．
6. CASMの出力をASMによってアセンブルした後につくられた.HEXファイルはCP/MのLOADコマンドを使用して，二進のファイルへ変換することはできないかわりに，DDTあるいは，SIDがそのファイルをメモリーへ読み込むために使用されなければならない．また，CP/M SAVEコマンドは，そのファイルを.CRLファイルとしてセーブするために使用しなければならない．CASMでエラーが発生したら，対応するASMの行の最後に"！"を付加する．ファイルをメモリーへ読み込むためにDDTあるいはSIDを使用した後，256バイトのブロックがいくつセーブされる必要があるのかを正確に知るためには，左はしにプリントされる値に基する．LOADが使用されない理由はCASMファイルの末尾でCRLファイル・ディレクトリを生成するためのコードを出力するからである．そしてTPAのベースへそのロケーションのカウンタをセットするために，"ORG"擬似操作を利用している．この性能を持つアウト・オブ・シーケンスのデータに出会ったとき，LOADコマンドは秘密のメッセージ"INVERTED LOAD ADDRESS"で打ち切る．CASMにディレクトリ

情報を新しいファイルへ書き出させ，この新しいディレクトリ・ファイルの末尾の前の出力全体を付加させるよりも，私はむしろSAVEコマンドを使用することをユーザーに要求したい．

7. CASM.SUBのサブミット・ファイルは，最後のSAVEコマンドを入れる場合を除いて，.CSMファイルを.CRLファイルへ変換するための全手順を行なうために使用される．例えば，“FOO.CSM”と呼ばれるファイルを変換するには，

```
submit casm foo
```

とすればよい．処理が完了したら，“SAVE”コマンドを入力する．

付録　F

BDS Cファイル I/O の手引き

F.1　はじめに

ファイルの入出力を行なうためにBDS Cで与えられているライブラリ関数は2つに分けられる．原始または低レベルのI/O関数，そしてバッファ付きI/O関数である．

アセンブリ言語の中で最高の性能を求めてコード化される原始関数は，すべてのファイルI/Oを行う低レベルのCP/M BDOSコールに対する拡張されたインターフェイスである．原始I/Oコールの間に転送されたデータの量は常に一つの全CP/M論理セクタ（128バイト）の倍数である．

Cで書かれるバッファ付き関数は，フィルター・タイプの応用のために特に適応されたバイト単位の連続するI/Oシステムを与える．これらは，目にみえないメカニズムがすべてのセクタ・バッファリングや実際のディスク転送を操作するので，ユーザーが最も手頃な量がどれほどであっても，その中でデータを読んだり，書き込んだりできるようにする．このようにバッファ付きI/O関数は，たいてい原始関数よりも取り扱いが便利であるが，それらは実行スピードやコードバッファ領域のメモリースペースの消費といった点ではかなり超過する．

原始I/O関数は，バッファ付き関数を形づくるので，まず私は原始I/Oを詳しく説明し，次にバッファ付き関数へといってみよう．

F.2　原始ファイルI/O関数

すべての原始I/O関数は，操作中のファイルを識別するためのファイル記述子の使用によって特徴づけられる．ファイル識別子fdは，小さな整数値であってファイルが開かれたり，作成されたりする時にファイルへ割り当てられる．そして閉じられるまではファイルと結合したままでいる．fdはopenかcreat関数のいずれかをコールすることによって得られる．これらの関数の使い方は，

```
fd=open (filename,mode);
fd=creat (filename);
```

"filename"は文字列か式のポインタでなければならない．

Openは，読み取りか書き込み，もしくはその両方のためにすでに存在しているファイル（たいていは，その中のデータをいくつか持っているファイル）を開くために使用される．creatは，読み書きのために新しいファイルを作成したり開いたりするのに使用される．相方の場合，fdは，うまくいったときに呼ばれたプログラムによって戻される．何かエラーがあったり，指定されたファイルが作成もしくは開いたりされない場合は，ERROR（－1）が戻され，error関数がファイルが開かれなかった原因をみつけ出すためにコールされる．

すべての他の原始関数は，ファイルが操作されるように指定するため，fdを必要とする（unlinkとrenameを除く，これらはファイルネーム・ポインタを必要とする）2つの非常に重要な原始I/O関数，readとwriteは，128バイトの論理セクタの倍数でデータをディスクへ，あるいはディスクから転送する．その典型的な使い方は，

```
i=read (fd, buffer, nsects);
j=write (fd2, buffer2, nsects2);
```

はじめのコールは，データの nsects 分のセクタを，その fd が指定されているファイルから，ロケーション buffer でのメモリーへと読み取ろうとする．2つめのコールはデータの nsects2 をロケーション buffer2 でのメモリーから，その fd が fd2 であるようなディスク・ファイルへ書き込もうとする．エラーが起こらなければ(まちがった fd が与えられたり，ファイルの末尾をすぎても読み取りが行なわれていたりするときのように)，read と write はすぐにディスク操作が行なわれるようにする．これは，原始 I/O とバッファ付き I/O の大きなちがいの一つである．原始関数はいつも行ないたいことが可能である限り，ファイル I/O にすぐ活動させるようにするが[22]バッファ付き関数はバッファがいっぱいになったとき(書き込みの間)，あるいは消耗してしまった(読みの間)ときに，ディスクアクセスをする．

原始 I/O のために開かれた各ファイルと結合している目にみえない“r/w ポインタ”がある．このポインタは次に続くセクタのトラックをファイルから読まれ，あるいはファイルへ書き込まれるためにキープする．ファイルが開かれたすぐ後に r/w ポインタは 0 に初期化される(ファイルの第1のセクタ)．それはうまく転送されたセクタのナンバーによって read や write のコールをつづけ，自動的に増加される．だから，省略時解釈によって，各データの転送は前の転送が終わった地点からピック・アップする．ファイルの r/w ポインタの値は tell 関数によって戻され，seek 関数を使用することによって修正される．

プログラムの中で原始 I/O の使い方を解説するには，ファイルのコピーをつくるために単一のユーティリティーをうち立ててみるとする．このユーティリティー（我々は“copy”と呼んでいる）のためにコマンド・フォーマットは次のようであるべきである．

22) ほとんどの CP/M のシステムでの原始ファイル I/O コールは，ディスク・ドライブのハードウェアはすぐに活動できるようにする．あるシステムでは BIOS がセクタのバファアリングをするが，各々のそしてすべての原始 I/O コールのために物理ディスクを動かす必要はない．

A> copy fillname newname <cr>

"copy"は"filename"によって指名されたファイルであり，"newname"と呼ばれるそれのコピーをつくる．これは上質のユーティリティーであるべきなので，うまくいかない場合には，十分なエラー診断が必要である（ディスク・スペースがなくなったり，マスター・ファイルがみつからなかったりした場合など）これは正しい数のパラメタがコマンド・ライン上にタイプされたことを確認する検査を含む．半分C/半分英語の擬似コード・フォームを用いて，プログラムを要約するのはときには便利なものである．これはフローチャートのようなものであるが，"条件や制御の箱"を使ったものではない．次のものはそのようなコピー・プログラムの要約である．

```
copy (file1, file2)
 {
        if (exactly 2 args weren't given)
                    complain and abort
        if (can't open file1)
                    complain and abort
        if (can't create file2)
                    complain and abort
        while (not end of file1)      {
                    Read a hunk from file1 and write it out to file2;
                    if (any error has ocurred)
                                complain and abort
        {
        close all files;
}
```

そして，コピー操作をするための実際のCプログラムは次のようになる．

```
# include <bdscio.h>       /* The standard header file              */
# define BUFSECTS 64       /* Buffer up to 64 sectors in memory     */

int fd1, fd2;              /* File descriptors for the two files    */
char buffer [BUFSECTS * SECSIZ] ;  /* The transfer buffer           */

main (argc, argv)
int argc;          /* Arg count          */
char ** argv;              /* Arg vector          /*
 {
        int oksects;       /* A temporary variable */

                           /* make sure exactly 2 args were given   */
        if (argc !=3)
                perror ( " Usage: A>copy file1 file2 <cr>\n " );

                           /* try to open 1st file; abort on error  */
        if ( (fd1=open (argv [1] ,0) )= =ERROR)
                perror ( " Can't open:%x\n " , argv [1] );

                           /* create 2nd file,abort on error: */
        if ( (fd2=creat(argv [2] ) )= =ERROR)
                perror ( " Can't create:%s\n " , argv [2] );

                           /* Now we're ready to move the data:     */
        while (oksects=read (fd1, buffer, BUFSECTS) ) {
                if (oksects= =ERROR)
```

```
                perror ( " Error reading:%s\n " , argv [1] );
            if (write (fd2, buffer, oksects)!=oksects)
                perror ( " Error; probably out of disk space\n " );
    }

                /* Copy is complete. Now close the files:   */
    close (fd1);
    if (close (fd2)= =ERROR)
            perror ( " Error closing %s\n " , argv [2] );
    printf ( " Copy complete\n " );
}

perror (format, arg)   /* print error message and abort          */
 {
    printf (format, arg);       /* print message     */
    fabort (fd2);               /* abort file operations */
    exit ( );        /* return to CP/M   */
}
```

さて，このプログラムをみてみよう．最初に宣言である．コピー処理に伴うファイルのファイル記述子が必要である．そしてディスク・ファイルのデータをバッファするための大きな配列をメモリーに割り付ける．バッファのサイズは，バッファされるセクタの数（プログラムの先頭で定義されているBUFSECTS）とセクタの大きさ（BDSCIO. Hの中で定義されているSECSIZ）を掛けた値である．

main関数においては，まず最初に正しい数のパラメタがコマンド行でタイプされたことを確認する．" argc "パラメタは，各々のmainプログラムに対してランタイム・パッケージによって自由に与えられていて，常に与えられてた

パラメタの数に1をプラスしたものと等しいので，それが3に等しいことを確認する（これは2つのパラメタが与えられた場合）．argc が3に等しくない場合，不服を申し立てそのプログラムを打ち切るために perror をコールする．perror はその引数が printf コールに対する最初の2つのパラメタであるかのように，それらを理解し，要求された printf コールを行ない，出力ファイル[23]での操作を打ち切り，コマンド・レベルに戻る．

argc テストを終えたら，次はファイルのオープンを試みるときである．次の命令文は読み取りのためにマスター・ファイルを開き，open によって戻されたファイル記述子を変数 " fd1 " へ割り当て，そして open がエラーを戻したなら，そのプログラムは打ち切られるようにする．これは，Cの式・評価子のおかげで一度にすべて行われる．一つの命令文の中でこういったたくさんのことが起こるのを見ることに慣れていない場合，注意深く調べるためにしばらく間を取りなさい．まず，open へのコールがなされ，open からのリターン値が変数 " fdl " へ割り当てられ，それからその値が ERROR であったかどうかを知るためにテストが行われる．その値が ERROR に等しくなかった場合，そのファイルは正しく開かれたので，コントロールは次の if 文へパスされる．でなければ，perror に対する適切なコールがその問題を診断し，そのプログラムを終結させる．出力ファイルの作成は，似たパターンをつづけ，ファイル作成を試みたがエラーを戻したという場合には，perror コールを用いて問題を診断する．

準備が全部できたら（いよいよ！）．データのコピーを始めるときである while ループを使用するごとに，我々は得られるだけのデータ（BUFFSECTS セクタ数まで）をマスターファイルからメモリーへ読み込む．read 関数は，うまく読まれたセクタの数を戻す．これは，0（ファイルの終りの条件を示している）から要求されたセクタの数（この場合，BUFFSECTS）までの範囲で，アクシデントがあったとき（ディスク・ドライブのドアーが開いてしまったり）には ERROR の値を戻す．

23）これは，ファイルが開かれる前にコールされると効果がない．例えば，まちがった数のパラメタが与えられてしまって，" argc ! = 3 "のテストが成り立ったときのように．

この値がどんなものであっても，後の試験のために“oksects”に割り当てられる．これがゼロに等しいような場合，while ループは退出される．さもなくば，我々はそのループに入り，読み込まれたデータを書き出そうとする．だが，我々は最初にとんでもないエラーが起こってないかを確認したい．だから，ERROR が read コールによって戻されたかどうかを見るためにチェックが行なわれる．もしそうであれば，中断する．なにもエラーがなければ我々はそのデータを出力ファイルへ書き出すために write をコールする．書きたいセクタの数を正確に書き込むのに成功しない場合は，それは適切なエラー・メッセージを表示し中断する（ほとんどの書き込みエラーはディスク・スペースがなくなることによって起こる）．write が成功した場合，ループの最初にもどり，もっとデータを読み込もうとする．このプロセスはすべてのデータを読み取り，書き込みされてしまうまで続く．このとき read 関数がゼロを戻したら，コントロールは while のループの手から離れる．

いったん while ループが離れてしまったら，最後に行うことは，ファイルを閉じることである．出力ファイルが正しく閉じたことを確認しプログラムは終了する．

F.3 バッファ付きファイル I/O 関数

先程のセクションで表わされた原始ファイル I/O 関数は，多量なデータを操作するようなときに最も有効である．前のファイル・コピーのプログラムは典型的な一つのアプリケーションである．原始ファイル I/O はユーザーが常にセクタという単位からみて考えるように要求する．これはファイルコピーの例におけるように，特殊な問題は持ってないが，ビットやランダムなサイズのデータを扱うときにごく少しだが複雑なものを加える必要がある．たとえば，テキスト・ラインとして知られるユニットを考えてみると，ASCII データの一行の長さは，1バイトから（空白ストリングの場合 NULL によってのみ表わされる）130バイトかあるいはもっとあるかもしれない．ディスクファイルへこれらのテキスト行をあるいはディスクファイルからこれらのテキスト行を読んだり，書いたりするのに便利な方法は，テキスト処理のアプリケーションにとって大変

便利なものである．理想的にいえば，我々はテキスト行のポインタと共にある種の記述子をパスして単一関数をコールしたい．そしてその関数にテキストの行を最後に書かれた行につづくファイルへ書き込みたい．また，一行がかかれるごとに時間を浪費するディスク・アクセスを防ぐためには我々の関数に，多くの行をバッファさせバッファが満たされると，すぐそれらの行全部をディスクへ書き込ませるとうまくいく．同様にファイルからメモリーへテキストの行を読み込む関数がなければならない．もし，ディスク動作が最小限におさえられるように目にみえないバッファがテキスト行を保つ関数によって処理される場合，それは非常にその性能を改善する．いま延べられた関数は事実，標準ライブラリの fputs や fgets である．これらはバッファ付き I/O 関数の 2 つの例である．

バッファ付き I/O の特徴は，論理的な I/O バッファと呼ばれる構造体を持っていることである．この構造体の中には，転送されるデータを記憶する大きな文字配列があり，またバッファのデータ配列部分の中で起こったことを見のがさずに覚えておくためにいくつか組み合わせたポインタや記述子がある．これらは原始 I/O 操作のためのファイルを識別するファイル記述子，次のバイトが書き込まれたり，あるいは読み取られたりする場所を教えるためのデータ配列のポインタ・カウンターが，バッファを再ロードしたり，書き出したりする必要があるようになる前に，データかスペース（読み取っているか，書き込んでいるかにかかわってくる）のいずれかのバイトが，どれほど残されるのかを教えるためのカウンター，それから最後に，ファイルが閉じられるときに事がうまく運ぶように入力あるいは出力のために使用されているかどうかといったようなことを記憶するバイト，これらのものを含んでいる．バッファ付き I/O 関数はまさに，原始ファイル I/O 関数がファイル記述子を使用するように操作されているファイルのための識別として，これらの I/O バッファに対してポインタを使用する．

単一バイトのデータや文字のために，実際のバッファ付き I/O を実行する 6 つの関数がある．他のバッファ付き I/O 関数（fputs や fgets のように）は，これらの 6 つの“中堅”関数を用いてそれらの仕事を行う．

ファイル読み取りのために，fopen, getc, fclose といった関数がある．Fopen は既存のディスク・ファイルを開くためにコールされ，そのファイルをユーザーの与えたI/Oバッファと結合させ，ファイルから受け取ったデータのためにそのバッファを初期化する．Getc はデータ配列が空であるとわかるときは，いつでもディスク・ファイルからデータ配列を必ず詰め替えて，バッファから単一バイト（文字）を取り出す．そして，物理的なファイルの終りに達したときに，特別なEOFの値(－1)を戻す．Fclose は，I/Oバッファと連合したファイルを閉じ，他のファイルを使用するためにバッファを開放する．

ファイル書き込みのためには，fcreat pufc，fflush，fclose といった関数がある(fclose は双方に成立する)．fcleat は新しいファイルをつくり，出力データのために連合したI/Oバッファ構造体を準備する．putc へのコールによって，一回につき1バイトのデータをバッファへ書き込む．putc コールによってバッファがいっぱいになったときは，いつでもそのバッファはディスクへ書き出され，他のデータを受け入れるためにリセットされる．データすべてがファイルへ書き込まれてしまったとき，fclose は連合したファイルを閉じることによって処理をし完結させる．出力ファイルのために fclose は，ディスク・ファイルが閉じられる前にディスク・ファイルに対して，まだ十分でないI/Oバッファの内容をダンプ・アウト（“フラッシュ”）するために，まず fflush を自動的にコールする．

実際にデータを読み書きする関数は，getc と putc である．fgets，fputs，fprintf などの関数は，getc と putc によってその読み書きを行なう．

BDSCIO. H ヘッダーファイルを調べれば，バッファリングに使用されるセクタの数は，8であるということに気付くであろう．この値は，異なるシステムにおける最適パフォーマンスのためにユーザーによって，変更されるということができる．たとえば，1024バイトの物理セクタディスク・フォーマットを持つCP/Mシステムで，BDS C を使用している場合，バッファ付きI/O関数によってなされたバッファリングの1024バイトはたぶん不必要であり，8セクタから1セクタまでのバッファリングを変えることによって，実行スピードで重要なロスをせずにごくわずかだがメモリーをセーブする．だが，8インチ標

準128バイトの物理セクタを実行するCP/Mシステムでは，デフォルトの1Kバッファリング案は本当に処理をスピードアップする．

最初に簡単な例をみてみよう．次のプログラムは一番左のマージンに行番号を付けてコンソールにテキスト・ファイルをプリント・アウトする．

```
/*
        PNUM. C:Program to print out a text file with
                automatic generation of line numbers.
*/

# include<bdscio.h>

main (argc,argv)
char ** argv;
 {
        char ibuf [BUFSIZ] ;           /* declare I/O buffer         */
        char linbuf [MAXLINE] ;        /* temporary line buffer      */
        int lineno;                    /* line number variabele      */

        if (arge !=2)  { /* make sure file was given   */
                printf ( " Usage: A>pnum filename<cr>\n " );
                exit ( );
        }

        if (fopen (argv [1] , ibuf)==ERROR)  {
                printf ( " Can't open %s\n " , argv [1] );
                exit ( );
        }
```

```
        lineno=1;                    /* initialize line number         */

        while (fgets (linbuf, ibuf))
                printf ( " %3d:%s ", lineno++, linbuf);

        fclose (ibuf);
}
```

ibuf は，fopen，getc，fclose 関数で利用される I/O バッファ領域である．BDSCIO. H の中で定義されたシンボル定数 BUFSIZ は，I/O バッファがどれ程のバイトを含んでなければならないかを教える．この値は上で示されたようにデータのバッファリングに必要なセクタの数によって変わる．

引数カウントをチェックし，バッファ付入力のために指定されたファイルを開いた後に，すべての本当の作業が単一の while 文の中で実行される．まず，fgets 関数は，ファイルからテキスト一行を読み取り，それを linbuf 文字配列の中へおく．ファイルの末尾に出会わないかぎり，fgets はゼロ以外の（真の）値を戻し，while 命令文のボディが実行される．そのボディーは printf への単一コールで成り立っていて，その中では現在の行番号をコロン，スペース，現在のテキスト行によって続けられ，プリント・アウトされる．lineno の値が使用された後に，それは次の繰り返しの準備のために（++演算子によって）増加される．

読み取りとプリントのサイクルは，fgets がゼロを戻すまで続き，その地点で wrile ループをぬけ出し，fclose をコールして処理を完結させる．

最後の例で，フィルターとして知られるある種のプログラムを紹介する．普通，フィルターは入力ファイルを読み取り，そこである種の変換を行い，新しい出力ファイルへその結果を書き出す．その変換は（C のコンパルのように）きわめて複雑であるかもしれない．あるいは入力テキスト・ファイルを大文字に変換するのと同じくらい些細なことかもしれない．近頃，印刷料はかなり高いので，時間節約のため，C のコンパイラーはとばし，大文字変換のフィルタ

ー・プログラムをみてみよう.

```
/*
        UCASE. C:Program to convert an arbitrary input text
                file to upper-case-only.
*/

# include<bdscio.h>

main (argc, argv)
char ** argv;
{
        char ibuf [BUFSIZ] , obuf [BUFSIZ] ;
        int c;

        if (arge !=3)  {
                printf ( " Usage: A>ucase file newfile <cr>\n " );
                exit ( );
        }
        if (fopen (argv [1] , ibuf)= =ERROR) {
                printf ( " Can't open %s\n " , argv [1] );
                exit ( );
        }
        if (fcreat (argv [2] , obuf)= =ERROR)  {
                printf ( " Can't create %s\n " , argv [2] );
                exit ( );
        }

        while ((c=getc (ibuf))!=EOF && c !=CPMEOF)  {
```

```
                if (putc (toupper (c), obuf)==ERROR)
                        printf ( " Write error; disk probably full\n " );
                        exit ( );
        }

        putc (CPMEOF, obuf);
        fclose (obuf);
        fclose (ibuf);
}
```

今回は処理されるのに2つのバッファ付きI/Oストリームがある．入力ファイルと出力ファイルである．最初の仕事は，正しい数のパラメタがコマンド行で与えられたかどうかをチェックすることである．この場合，我々は2つのパラメタを必要とする．既存する入力ファイルの名前と，作成される出力ファイルの名前である．それから，バッファ付きI/Oのための2つのファイルを開き，作成するためにfopenとfcreatがコールされる．それがうまくいったなら，mainループに入り，楽しいことが始まる．

ループの各々の繰り返しのときに，単一のバイトが入力ファイルから取り出され2つのテキスト・ファイルの終りの値，EOFとCPMEOFと比較される．普通テキスト・ファイルの中の最後のものは，CPMEOF（コントロールZ）の記号である．しかし，そのファイルが偶然，セクタ境界線上できっちりと終わった場合，あるテキスト・エディタはファイルの終りにCPMEOF記号を置くことをしない．この場合，CPMEOFはみつからないので，物理的なファイルの終りの値(EOF)を検知しなければならない．この処理は多少複雑である．getcによって戻されたEOFの値は－1であり，これは16ビットの値として表わされなければならない．BDS Cでのchar変数は負の値を持つことができないので，変数" c "をcharのかわりにintとして宣言する．もし，それがcharとして宣言されると次式

c＝getc (ibuf)

は,char の値になるので,EOF と等しくなることはあり得ない.そういった場合に, getc が EOF を戻した場合, " c "は255に等しいままで終わる(値 EOF の低位8ビットの char 解釈). このように" c "は EOF の比較が意味のあるようにするために, int として宣言される. これは具合がよいものでない. というのは" c "はここでは文字を保有するために使用されているからで, それが文字変数として宣言されるようにするとよい. 実際にそれを行う方法がある. その比較の中で EOF が255へ変更されたなら, " c "は char として宣言されなければならないし, そのプログラムは [実際の16進の FF (10進の255) バイトに入力ファイルの中で出くわすときを除けば] 作動する. さて, ユーザーの平均的なテキスト・ファイルの中に16進の FF バイトがないと想定するのは, とても安全なかけであるが例外はある. また, フィルターがテキスト・ファイルのためだけに書き込まれうるという規則はない. 二進ファイルをアンロードし, インテル・フォーマットの HEX ファイルを作成するプログラムを考えてみなさい. 最初の16進の FF に出くわしたとき, 処理を停止したいと思うだろうが, ノーである. もともとの方法は明らかに最も一般的なものである.

ファイルの終りに出くわさなかったと決定した後, while 命令文のボディーが実行される. ここで我々は, getc からえた文字を大文字に変換するために toupper を使用する. それから, 出力ファイルへ出てきたバイトを書き出すため pufc を使用する. 適切であるかどうかをみるのにエラーをチェックする. putc が ERROR を戻す場合, そのプログラムは終結する.

ファイルの終りの条件が検知されると, すぐ我々は出力ファイルを終結するために, 最後の CPMEOF(コントロールZ)記号を書き出す. このプログラムの手法では, CPMEOF は入力ファイルが CPMEOF で終わっても終わらなくても, 出力ファイルへ付加される. 最後に fclose は入力／出力ファイルを閉じるために使用される.

バッファ付き I/O の使い方について大がかりな例をみるなら, CASM. C をみなさい. また, 何度かファイル BDSCIO. H, STDLIB1. C, STDLIB2. C を

調べてみなさい．これらは，バッファ付きI/O関数全部のソースを含んでいるSTDLIB1.Cは，バッファ付きI/Oライブラリの一般的なバイト単位の処理の部分を含んでいるし，STDLIB2.Cはライン単位の処理やフォーマット変換関数を含んでいる．

付録　G

BDS C コンソール I/O のひっかかりやすい点の解説，実例

G.1　はじめに

この文書の中で私は CP/M コンソールの入力／出力のメカニズムの背後にある不可思議な部分を取り除き，BDS C プログラムからどのようにしてそのメカニズムを最良に利用するかを示そうと思う．

ここで一番重要なポイントは，CP/M の BIOS と BDOS によって直接，コンソール入力，コンソール出力をするためにどのようにして bios と bdos のライブラリ関数を使用するかという点である．標準ライブラリの中で与えられた getchar/putcher 関数を使用するかわりに，コンソール I/O のために CP/M の BIOS へと直接行く一つの理由は，CP/M BDOS と getchar/putchar 関数（これはそれらの仕事をなすために BDOS コールを使用する）の両方によってある ASCII 文字の予期せぬ妨害をさけるためである．ある適切なアプリケーションはテレコミュニケーションのプログラムやゲームそして標準 getchar や putcher 関数が与えるのよりも，もっと直接的なコントロールをコンソールに求めているプログラムなどである．

BDS C（たとえばユーザーズ・ガイド）のための主な文書が，数年前に準備されたとき，私はどのようにして BDOS と BIOS を通して直接コンソール I/O を実行するために，bdos と bios ライブラリ関数が使用されるのかを，すべてのユーザーが理解するのかというつまらない憶測をしていた．そういった目的のための bios と bdos 関数の使い方は，CP/M システムのプログラマーにとって

は自明のことであり，アセンブリあるいはマシン言語のプログラムからCP/Mコンソールをドライブしたことのあるプログラマにとってもそうである．

G.2 基本コンソールインターフェイス

コンソールI/Oの間で何が起こるのか，あるいはそれをどうコントロールするのかをみてみよう．

コンソールコントロールのソフトウェアの最低レベル（最も簡単な）はCP/MのBIOS（基本入力／出力システム）の中にある．コンソールの文字を読み取ったり，書き込んだりするのを処理する3つのサブルーチンがある．：CONST（コンソールステータスをチェックする）CONIN（文字がコンソールからタイプされるのを待つ），CONOUT（コンソールに文字を出力）アセンブリ言語レベルから，これらのサブルーチンの位置指定をする方法はむしろ複雑である．だから，BDS Cライブラリは Cプログラムから BIOSサブルーチンをアクセスするのをかんたんにする bios関数を含んでいる．

BIOSのベクトル2，3，4はコンソール装置と直接コミュニケートするために使用される．式bios(2)は，biosの中でCONSTサブルーチンへの値を戻す．あるいはそうでない場合はゼロを戻す．bios(2)がある文字が準備されていると示した後にその文字を実際に読むためには，あるいは文字が準備でき，それを読み取るまで待つためにはCONINサブルーチンをコールし，コンソールから文字を戻すのにbios(3)を使用しなさい．コンソールへ文字Cを直接書き込むためには，CONOUTをコールするのにbios(4, c)使用する．だが，BIOSは，Cプログラムが単一の“ニューライン”記号（‘＼n’）によってキャリッジリターン／ラインフィードの組み合わせを表わしていることに気付いていない．bios(4, ‘＼n’)のコールは，単一のラインフィード記号(ASCIIの十進値10)だけがキャリッジ・リターンなしにコンソール上にプリントされる．直接コンソールI/Oを使用しているとき，コンソール出力の最初のけたにゆくためにCONOUTサブルーチンへキャリッジ・リターン（‘＼n’）とニューライン（‘＼n’）の両方を送る．

そういった手順は次のようになる．

```
bios (4, '\r'); /* send carriage-return to CONOUT */
bios (4, '\r'); /* send linefeed to CONOUT */
```

すべてのコンソール I/O は，3つの BIOS サブルーチンによって実行されるが，全コントロールがコンソール装置上に要求されているときこれが異なる CP/M システムの間でプログラムをうごかす唯一のアプローチの仕方であることを確認している[24]．

G.3　BDOS とその複雑さ

コンソール I/O の次に高いレベルのインターフェイスは，BDOS（基本ディスク OS)である．コンソールでのインターフェイスを行うために，3つの基本 BIOS サブルーチンがあるのと同じように，よく似た作業を行うためによく似た3つの，しかし，"より高いレベル" の BDOS がある．これらの BDOS ファンクションは，BIOS のものとは明らかに違う BDOS 自身のコード・ナンバーを持っている．コンソールから文字を得るための Console Input（BDOS ファンクション1)，コンソールへ文字を書き込むための Console Output (BDOS ファンクション2)，コンソール入力から手に入る文字があるかどうかを決定する Get Console Status（BDOS ファンクション11)

標準Cライブラリ関数 getchar と putchar がコールされるときはいつでも，それらは BDOS コールを用いてその作業を行う．これは次に BIOS コールを通して操作を行い，ひどい混乱へと導く．BDOS の操作はユーザーがまだ十分気付いていないようなことをユーザーのためにあらゆる種類にわたって行ってくれる．たとえば BDOS コンソール出力コールでコントロールSの記号がコンソールから入力されると，コントロールがもとの出力コールから戻る前に別の記

24) たとえそうでも，どんな種類の端末が別のシステムによって使用されているのかを知る方法はない—だから "本当にポータブルな" ソフトウェアが使用されている表示端末の種類について仮定するか（それがカーソルのアドレッサブルかそうでないか，そのカーソルをどのようにしてアドレスするかなど)，あるいは，最後のユーザーがそのシステムに偶然リンクさせたのがどんなタイプの端末であっても，フィットさせるために自己修正用の準備を含んでいるかのいずれかである．

号が入力でタイプされるまでストップする．ユーザーのCプログラムへその特徴をコードする必要もなく長いプリントアウトをストップしたり，スタートしたりする能力をユーザーが欲している場合，これはよいかもしれない．しかし，コントロール－Sを含むコンソール上にタイプされた各々の文字をみる必要がある場合，大きな問題を引き起こす．BDOSがこのことをどのようにして行うか．ということに関して，ほんの少し知っていれば，いくつかのおもしろいことがわかる．BDOSはコンソール入力でコントロールSを検知できなければならないので，それは文字がタイプされてしまったと見るときは，いつでもコンソールを読まなければならない．文字がコントロールSのように特別な処理を必要とするものの中にあるのでなければ，次の“Console Input”コールが何事も起こらなかったのごとく文字を戻すために，その文字はBDOSに対して内部のどこかでセーブされなければならない．“Get Console Status”（“console Input”に対して何かがなされる前）に対してユーザーが行った続いておこるコールが文字が有効であることを示しているのをBDOSは確認しなければならない．これはBDOSのコールが文字が有効であると言うかもしれないような条件へ達する．しかし，符号するBIOSコールは物理的でない．というのは文字は先のBIOSとの相互作用の間にBDOSによって，すでに取り出されているからである．

もし，こういったことすべてが混乱しているような感じがするなら，私が何が起こっているのかを把握できるようになる前には，CP/Mやコンパイラの初期の頃のバージョンと取り組んで数カ月もかかったということを心にとめておきなさい．getcharとputcharのライブラリ・バージョンはユーザーがコンソール上に絶対的な直接のコントロールを必要としないような環境のためにデザインされた．BDOSはすでにいくつかよいことを行うので（コントロールSの処理のように），いくつか追加の特徴を延べることにする．出力時に‘＼n’記号をCR-LFの組み合わせに自動的に変換する（長い，あるいは無限の望まざるプリントアウトがそのマシーンをリセットせずにストップされるためにコンソール入力操作が行われてないときでも）コントロールCが入力時に検知されるとき，自動的にプログラムが終結する．入力時にキャリッジ・リターン記号が‘＼

n’へ自動的に変換される．ある初期の頃のユーザーは putchar をコントロールCから免除されるようにしたいといったので，彼のために私は putch ライブラリ関数をつけ加えた．これはコンソールでタイプされたときにコントロールCがそのプログラムをストップしないということ以外は，putchar と同じように作動する．その少し後になって CP/M が物理的コンソール入力をサンプリングするのを防がなければならないとき，putchar も putch の両方が妥当でないということが明らかになった．この地点で私は，bios 関数をつけ加えた．それでユーザーは BIOS を通して，直接 I/O をできるようになり，全体に無益な記号食いの BDOS をバイパスする．

私は始めの頃にいくつかの例をあげるといったのでそちらに移ってみよう．まず最初に bios 関数によって，3つの基本的なコンソール操作を行う．非常に基本的な関数がセットになっており，特別な変換もさまたげも全くない．たとえば，‘＼n’→ CR-LF のようなものはない．

```
/*
        Ultra-raw console I/O functions:
*/

getchar ( )            /* get a character from the console */
{
        return bios (3);
}

kbhit ( )              /* return true (non-zero) if a character is ready */
{
        return bios (2);
}
```

```
putchar (c)            /* write the character c to the consol */
char c;
 {
        bios (4, c);
}
```

これらの超原始的な関数は，BIOSのコンソールサブルーチンに対して直接のアクセスを与える以外は何もしない．DEFF2.CRL（これはついでながらアセンブリ言語で書かれており，DEFF2A.CSMのソースフォームの中にある）の中で与えられた標準バージョンのかわりに，それらを使用するためには，単にそれらを（あるいは，お好みのバージョン）を含んでいるCのソース・ファイルを作成し，ファイルをコンパイルし，ユーザーのプログラムと出てきたCRLファイルとをリンクしなさい．

さて，コンソールI/O関数のカスタマイズされたバージョンに取り組むことができるような，もっと洗練されたゲームを考えてみよう．手始めに，前に延べたライブラリ・バージョンのようなニューライン変換を行う直接コンソールI/O関数のセットをデザインしてみよう．そしてコントロールCで実行を打ち切りなさい．しかし，コントロールSの規約を無視しなさい．そして，出力の間にタイプされた文字がコントロールC以外ならそれを捨てなさい．コントロールCならば我々がここで必要としていることは，上記の概略の間とそれプラス，次の条件を操作するためのコードである．

a）出力時に単一の‘＼n’記号をCRとLFへ変換する．

b）入力時にCRをニューライン（‘＼n’）にコントロールZを－1に変換する．

c）コンソールへ入力を自動的にエコーする．

d）入力と出力の両方のときにコントロールCでリブートする．

次にあげているのは，やや意地悪であるが，

```
/*
        Vanilla console I/O functions without going through BDOS:
        Note that ' kbhit ' would be the same as the preceding
        ultra-raw version)
*/

#define CTRL_C 0x03    /* control-C */
#difine CPMEOF 0x1a    /* End of File signal (control-Z) */

getchar ( )            /* get a charcter, hairy version */
{
        char c;
        if ((c=bios (3))==CTRL_C) bios (1);   /* on Ctl-C, reboot */
        if (c==CPMEOF) return -1;             /* turn Ctl-Z to -1 */
        if (c== '\r') {                       /* if CR typed, then     */
                putchar ('\r');               /* echo a CR first, and set */
                c= '\n';                      /* up to echo a LF also */
        }                                     /* and return a '/n' */
        putchar (c);                          /* echo the char */
        return c;                             /* and return it */
}

putchar (c)        /* output a character, hairy version */
char c;
{
        bios (4, c);                  /* first output the given char */
        if (c== '\n')                 /* if it is a newline, */
              bios (4, '\r'));        /*    then output a CR also */
```

```
        if (kbhit ( )&& bios (3)= =CTRL _ C)     /* if Ctl-C typed, */
            bios (1);                            /* then reboot  */
}                               /* else ignore the input completely */
```

さて，もしあなたがコントロールSの処理とプッシュ・バックの特徴（この2つは実際によく関連している．というのはユーザーは，出力時に検知されるかもしれないコントロールSを除いて，何でもプッシュ・バックできなければならないから）をつけ加えたいのなら，関数の最後のセットへ外部の"状態"を付け加えることができ，コンソール入力でみることを覚えておくことができた．けれどこれが行われると，ユーザーが行ったことはもとの標準ライブラリ・バージョンgetcharとputchar（これはBDOSを使用する）とほとんど同じ関数になる．ユーザーはまたこれらを使用したにすぎないのかもしれない．

いままで私が延べてきたすべてのことは，BIOSの見地にたってのことであった．そしてすべてのCP/Mシステムに対して同様に与えている．運悪く，BIOSのサポートしていない実時間の対話型の操作を書くために，しばしば必要とされるコンソール操作が一つあって，CP/Mでそれを実行するための方法がない．欠けているものはコンソールが出力文字を受け入れる準備のできている場合に，BIOSを求める方法である．これが欠けているために起こる問題の一つの例はサンプルプログラムRALLY.C（BDS Cユーザーズ・グループから手に入る）でみられる．そこではそのプログラムはどんな瞬間でもキーボードから入力を読むことができなければならない．そしてそこに送られたデータの量がそれ以上の記号を拒否させ，そして記号が送られるようになるまでプログラムのコールを閉鎖させるようなときに，端末を待つのを拘束するようになる余裕がもてない．コンソールへ記号を送る唯一のしかるべき方法は，CONOUT BIOSコールを通して与えられており，そういったコールはどんなときでも，許容以上の長さでプログラムをタイ・アップするかもしれない．唯一の頼みのつなは，CONOUTを完全にバイパスし，より洗練されうるCの中でカスタマイズされた出力ルーチンを構成することである．これはオブジェクトコードの可搬性を犠牲にして，RALLY.Cの中で行われる．ユーザーは各々，独特のポート・

ナンバー・ビット・ポジション・ユーザーのコンソールをコントロールするI/Oハードウェアの極性を定義するために，ヘッダーファイルを構成しなければならない．もし，BIOSがコンソール出力ステータスをテストするために小さなサブルーチンを一つ以上含んでいさえすれば，それはより簡単だったのかもしれない．しかし人生というのはときにきびしいものである．

私がこれがCP/MコンソールI/Oインターフェイスのあいまいなふるまいのいくつかを正しく理解するのに役立つことを望んでいる．getchar, putcharなどのライブラリ・バージョンがどんな具合であるかということに関する実情を知るために，実際に実行させたりDEFF2A.CSMの中のソースをみなさい．そして，もしコンソールを用いて何かしたかったり，この文書をみてもどんなものかをはかり知ることのできなかったりしたなら，私はいつでも御相談に応じます（少なくとも私が電話のすぐ近くにいるときは）．

G.4　CIO関数ライブラリ

CIOと名付けられた新しいユーティリティー・パッケージ（CIO.CSMソース・フォームの中に含まれている）は，トータルな直接コンソールI/Oコントロールを要求するアプリケーションの中で使用するためのものである．

付録　H

浮動小数点・関数パッケージ

ボブ・マシアス

H.1　はじめに

浮動小数点のパッケージの構成は：

1）FLOAT.C　　　:Cで書かれた関数

2）FP　　　　　　:ワーク・ホース機能（DEFF2.CRL の中）

3）FLOATSUM.C:サンプル・プログラム

ここにあげるのは，どのようにして動作するかということである．使用したいと思うすべての浮動小数点数のためには，ユーザーは5つのエレメントを持つ文字配列を宣言しなければならない．それから，関数コールの中でそれを指定する必要のあるときは，いつでも配列のポインタをパスしなさい．ボブの関数はその引数がそういった文字配列に対するポインタであるように思っている．

4つの基本演算機能は:fpadd, fpsub, fpmul, fpdivでこれらは各々3つの引数を取る．それらは，その結果が出るような5文字の配列に対するポインタと2つの演算子（浮動小数点のオペランドを表わしている5文字の配列に対するポインタ）

結果は悪影響を持たない引数のいずれかに置かれる．

たとえば，操作：

fpmult (foo, foo, foo);は，foo と foo を乗算し，結果を foo の中に置く．

ユーザーが要求している値に対して，浮動小数点の文字配列を初期化し，人間の読めるフォームの中でその値をプリント・アウトするために，次の関数が含まれている．

－ftoa は，浮動小数点の値を ASCII の文字列（ユーザーが"puts"を用いてプリント・アウトできる）へ変換する．

注：　この関数の明白な使用は FLOAT.C の中で特別にカスタマイズされた printf ファシリティーを使用するときは必要でない．

－　atof は，ASCII の文字列（NULL で終結された）を浮動小数点の値に変換する．

－　itof は，整数を浮動小数点の値に変換する．

H.2　詳しい関数要約

次の関数は，BDS C のユーザーが実数をアクセスし，操作できるようにするためのものである．各々の実数は，5バイトの文字配列に割り当てられる（char fpno[5]）．最初の4バイトは，LSB から始まる仮数部で5番目のバイトは指数である．

fpcomp (op1, op2)

char op1 [5] , op2 [5] ;は次の値を戻す．

もし	op1 > op2	ならば	1
もし	op1 <op2	ならば	－1
もし	op1 ＝op2	ならば	0

ほとんどの浮動小数点パッケージに関していえば，浮動小数点数を取り扱うときは同等性を比較するのはよいことではない．

```
char * fpadd (result, op1, op2)
char result [5] , op1 [5] , op2 [5] ;
```

は，op1＋op2 の結果を result に記憶する．op1 と op2 は浮動小数点の値でなければならない．result の先頭のポインタを戻す．

```
char * fpsub (result, op1, op2)
char result [5] , op1 [5] , op2 [5] ;
```

は，op1－op2 の結果を result に記憶する．op1 と op2 は浮動小数点の値でなければならない．そして result の先頭のポインタを戻す．

```
char * fpmult (result, op1, op2)
char result [5] , op1 [5] , op2 [5] ;
```

は，op1 * op2 の結果を result に記憶する．op1 と op2 は浮動小数点の値でなければならない．そして result の先頭のポインタを戻す．

```
char * fpdiv (result, op1, op2)
char result [5] , op1 [5] , op2 [5] ;
```

は，op1/op2 の結果を result に記憶する．op1 と op2 は浮動小数点の値でなければならない．ゼロによる割り算の結果は 0 である．result の先頭のポインタを戻す．

```
char * atof (op1, s1)
char op1 [5] , * s1;
```

は，ASCII の文字列 s1 を浮動小数点の値へ変換する．そして，op1 の中にその

結果を記憶する．この関数は先行するホワイト・スペースは無視されるが，値の中にホワイト・スペースを含むことはできない．次のものは正しい例である．" 2 "，" 2202222222283.333 "，" 2.71828e−9 "，" 334，3333E32 "．" 3443.33 E10 " は，不当である．なぜならば，スペースが含まれているからである．指数の値は：−38から＋38の範囲内でなければならない．op1 の先頭ポインタが戻される．

```
char * ftoa (s1, op1)
char * s1, op1 [5] ;
```

は，浮動小数点の値 op1 を ASCII の文字列に変換し，s1 から記憶する．7 桁の精度を持つ具体的な表記法で書式化される．その文字列は NULL によって終結する．そして，s1 の先頭ポインタを戻す．

```
char * itof (op1, n)
char op1 [5] ;
int n;
```

浮動小数点の値 op1 を整数 n の値へセットする．n は符号付き整数であると想定される．

H.3　一般的な注意

浮動小数点操作は，単純な式ではなく関数コールとして考えなければならない．浮動小数点パッケージの能力によって，コンパイラの能力を混乱させないために，特別な注意を払わなければならない．浮動小数点の値に初期値を与えるためには，たとえば次のような命令文を使用することはできない．

```
char fpno [5]
fpno = " 2.236 ";
```

そのかわりに

```
char fpno [5]
atof (fpno, " 2.236 ");
```

と命令しなければならない．さらに，" ival " と呼ばれる整数変数の値へ浮動小数点の値をセットしたい場合は，

```
char fpno [5] ;
int ival;
......
fpno = ival;
```

とやっても動作しない．最後の行を

itof (fpno, ival)；に変えなければならない．

もっと例をあげてみよう．

次のものは，100.2と−7.99を加えた結果を5文字の配列aに記憶する．

```
fpadd (a, atof (b, " 100.2 "),   atof (c, " −7.99 "));
```

（bとcも5文字の配列でなければならない．）

次のものは，aに1を加えない．なぜならば，op1とop2は浮動小数点の値(実際は文字に対するポインタ）でなければならない．

```
fpadd (a, a, 1); /* bad use of " fpadd "  */
```

上記の関数は，全部Cで書かれているが，それらのほとんどは本当にいやな作業を全部行うために，fpと呼ばれる単一のワークホース関数をコールする．この関数は，DEFF2. CRLに含まれている．これは，そのパッケージのマシー

ン・コード化された部分である．fP 関数のためのソース・コードは BDS C のユーザーズ・グループから手に入れる．あるいは，SASD (Self-Addressed, stamped 8" DISK:マスターディスケットのこと)を BD ソフトウェアに送りなさい．

付録　I

BDS Cのための倍精度整数パッケージ

ロブ・ショスタック

1982年 8月

I.1　はじめに

このパッケージは，ボブ・マシアス氏の浮動小数点パッケージと同じような精神でBDS Cに倍精度(32ビット)符号付き整数の能力をつけ加える．加減乗除とmodulusのルーチンや比較，割り当てなどいろいろな機能が含まれている．

倍精度整数は，4つの文字の配列に記憶される．倍精度整数xは，

char x [4] ;と宣言される.

内部の表現は，2つの補数フォームになり，配列の第1バイトが符号(MSB)である．しかし，ほとんどの目的では内部表現にたずさわる必要はない．

倍精度整数を操作するルーチンのほとんどは，3つの引数を必要とする．最初の引数は結果が記憶されることになる場所のポインタで，他の二つはオペランドである．たとえば，次の倍精度整数x，y，z（全部char［4］として宣言される）を用いた．

ladd (z, x, y)

は，xとyの合計を計算し，zへそれを記憶する．これはそのコールの値として戻される．その結果の引数はオペランドの引数のうちの1つ（あるいはその両方）と同じものであってもよい（たとえば，ladd (x, x, x) は"正しいこと"を行う）．

パッケージは，ある部分はCで，そしてある部分は（手っ取り早くつめこむため）8080アセンブリ言語で書かれている．それを利用するには，単にLONG.CRLをユーザーのプログラムにリンクすればよい．各ルーチンについての記述は下にあげられている．

```
itol (l,i)
char l [4] ;
int i;
```

は，16ビットの整数 i を倍精度整数の値として l へ記憶させる．そして l を戻す．

```
atol (l, s)
char l [4] ;
char * s;
```

は，Asciiの文字列 s を倍精度整数 l へ記憶させ，l を戻す．s の一般的なフォームは十進数字の文字列でマイナス・サイン（なくてもよい）によって先行され，数字以外のもので終結する．

```
Itoa (s, l)
char * s;
char l [4] ;
```

倍精度整数の値 l を Ascii の文字列にし，s に記憶しそして s を戻す．l が負

の数である場合，マイナス・サインが先行する．Ascii 数字は，NULL で終結される．s はその変換をうけ入れるのに十分な大きさでなければならない．

```
ladd (r, op1, op2)
char r [4] ;
```

は，倍精度整数 op1 と op2 の合計を r の中に記憶させ，r を戻す．op1 と op2 は r に使用することができる．

```
lsub (r, op1, op2)
char r [4] ;
char op1 [4] , op2 [4] ;
```

は，ladd と同様であるが，op1-op2 を計算する．

```
lmul (r, op1, op2)
char r [4] ;
char op1 [4] , op2 [4] ;
```

は，ladd と同様であるが，op1 * op2 を計算する．

```
ldiv (r, op1, op2)
char r [4] ;
char op1 [4] , op2 [4] ;
```

は，ladd と同様であるが，倍精度整数の商 op1/op2 を計算する．op2 がゼロなら結果もゼロとなる．

```
lmod (r, op1, op2)
```

```
char r [4] ;
char op1 [4] , op2 [4] ;
```

は，ladd と同様であるが，op1　mod　op2 を計算する．op2 がゼロなら結果もゼロとなる．

```
lcomp (op1, op2)
char op1 [4] , op2 [4] ;
```

倍精度整数 op1 と op2 を比較して，1，0，－1のうちの一つ(ふつうの整数)を戻す．戻す値はそれぞれ (op1> op2)，(op1＝＝op2)，(op1 <op2) の比較に対応する．

```
lassign (dest, source)
char source [4] , dest [4] ;
```

は，倍精度整数 source を倍精度整数 dest へ割り当て dest に対するポインタを戻す．

```
ltou (l)
charl [4] ;
```

は，倍精度整数 l を符号なしの整数に変換する．(切り捨てによって)

```
utol (l, u)
char l [4] ;
unsigned u;
```

は，符号なしの整数を倍精度整数の値に変換し，l に記憶させ，l を戻す．

1.2 内部動作

上記のルーチンの中の作業のほとんどは，long と呼ばれる単一8080アセンブリ言語の関数によって，つまり，ファイル LONG. CSM（ユーザーズ・グループから手に入る）の中に含まれるソースによってなされる．そのパッケージの残りは LONG. C の中にある．プリミティブのほとんどは，単に long をコールし，それにファンクションコード（どんな操作が行なわれるのかを教える）を操作される引数と共にパスする．

ファイル LONG. CRL は，LONG. C の中で与えられたコンパイル済みの関数を含んでおり，DEFF2. CRL はワークホース関数 long を含んでいる．

付録　J

TELEDIT 遠隔通信プログラムおよびミニスクリーンエディタ V1.1

ニゲル・ハリソン

"teledit"はオリジナル BDS Telnet プログラム，ワード・クリスティンソンの MODEM プログラム，その MODEM7 系のもの，XMODEM と呼ばれるCバージョンおよびニゲルのミニスクリーンエディーをつなぐものである．実際，このプログラムは細かく分けられるまで，しばらくは CDC 110 マイクロシステム上の基本エディタとして使われていた．このプログラムは，最終的に必要なモデムプログラムではないかもしれないが，他のどんなものよりも多くのルートをもっている．"TELEDIT"が，BDS C のディスクに含まれていなければ，BDS C ユーザーズグループから手に入れることができる．

Teledit は，通信プログラムでファイルを送ったり，他のシステムとつないだり，ASCII ターミナルのネットワークとして使うことができる．また，リモートシステムで，テキストを入れている間に，ラインを操作することができる簡単なエディタがついている．ファイル転送モードでは2進でもテキストファイルの状態でも受信送信が可能である．メニューからのモード選択は以下の通りである．

T：ターミナルモード……テキストは集積しない．

Teleditは ASCII ターミナルのように動く．8ビットキャラクタで受信送信される．パリティビットのチェック，挿入または移動はされない．SPECIAL キャラクタをタイプすることで，選択メニューにもどることができる．

SPECIAL キャラクタは，コントロールシフトのアプアロー(↑)に設定してあり，誤って打ってしまうことのないようにしている．もし，SPECIAL キャラクタを変更したければ，#define SPECIAL…で Teledit を再コンパイルすればよい．

X：ターミナルモード―テキスト集積をする．

モードは上と同様であるが，どんなキャラクタでも，通信リンクに受信されたものはテキストバッファにセーブされる．タブ，改行，書式送りのキャラクタもバッファ内に配置される．他のキャラクタは捨てられる．ターミナルモード中にエディタを呼ぶときはコントロールキーを押したまま，“E”をタイプすればよい．

Xモードは集積されたテキストに使われたファイル名を即座に出す．500行のテキストが入れられたあとで，テキストバッファにセーブされ，余分なラインが入ると，コンソールがアラーム音を出す．この状態になったとき，ユーザーは都合のよいところで，リモートステーションの通信を一時中止，下記にあるように，集められたテキストをディスク上にセーブする．

G：トグルエコーモード（エコーのセット）

もし，ユーザーが，全二モードで通信中リモートステーションからエコーを受信しているときが，半二重モードにあるときはトグルを入れるべきではない．このオプションを使って Teledit を走らせている他の人と話をするには，ファイルを送って，次に送られるファイルを持つ人に知らせる．

E：エディトテキスト集

メニューディスプレイから，エディタに入る．これは，テキスト集が入れられて，テキスト集ファイルがオープンされた状態で，X：ターミナルモードになるまで操作できない．エディタコマンドは，次に延べるとおりである．

F：テキスト集をバッファからファイルにセーブする．

テキスト集積モードで，バッファに集められたテキストをセーブする．ファイルをクローズしてはいけない．

U：CP/M ユーザエリアの選択

これはユーザエリアを持つユーザのためのもので，持っていなければ無視してよい．

V：CP/M ロジカルドライブ

使用可能なディスクドライブを選ぶ．選んだドライブは現在つながれているディスクとなる．

D：現在使用しているドライブとユーザ領域のディレクトリのプリント

現在のところのディレクトリは，UまたはVコマンドにより選ぶことができる．

S：ファイル送信，MODEM プロトコル

送りたいファイル名を打ち受信者からの“synch up”を待つ．
受信者が，この teledit と同じ MODEM プロトコルを使うプログラムを使用していなければならない．
操作終了後，メニューにもどる．

R：ファイル受信，MODEMプロトコル

受信したいファイル名を打ち，送信者から送信されてくるのを待つ．送信者はTeleditか同じMODEMプロトコルを使ったプログラムを使用していなければならない．

Q：終了

作業を止めてコマンドレベルにもどる．もし，Xモードでテキストファイルが集められていれば，セーブしたいかどうかを聞いてくる．

SPECIAL

SPECIALキャラクタを通信ラインに必要であるかぎり送る．SPECIALキャラクタはTELED.Cソースファイルの先頭で#difineステートメントによりコンパイル時に定義される．

スクリーンエディタ

エディタは，このモードに入ったときに“＊”を返してくる．現在の行の場所はプロンプトのすぐ下であることがある．エディタコマンドは下記のとおりである．

A　(append)ファイルに付けたす．現在の行の前に挿入することができる．

B　ファイルの始まりにもどって，そのページを画面上に出す．現在の行がテキストファイルの最初の行になる．

F　“F”のあとにつけられたパターンを含む行を捜し出す．見つかれば，その行が現在の行になる．前の方に向かって捜して行き1回りする．

I　挿入モードに入る．このモードをぬけるには，コントロールZを打ちリターンキーを打つ．挿入モードから

	ぬけるときは，行の先頭に入ったときでなくてはならない．そうでなければZは無視される．
K	現在の行を消す．
nK	n行消す．nは10進数．
L <pattern>	"L"のあとにあるパターンが行の先頭にある行を捜す．
O	〈コントロール〉Zが打たれるまでテキストの行上に書くことができる．〈コントロール〉Zは，新しい行の始めに打たれなければいけない．
P	テキストの次のページに行く．
−P	テキストの前のページに行く．
Q	エディタ終了
Sn	スクリーンサイズをn行にセットする．何もしなければ24行ディスプレイに22が定めてある．25行コンソールであればS23を使い，コントロールデータ110のときはS28を使う．
Z	ファイルのZee (end)終りに行く．最終行のページを見たいときなど，これで終りに行き−Pで見ることができる．
n <cr>	ファイル上でのn行前に進む．
<cr>	ファイル上で1行前に進む．
−n <cr>	ファイル上でn行後ろにもどる．
スペースバー	ファイル上で1行後ろにもどる．
#	ファイル中のテキスト行の数をプリントする．

移植

Teleditは，下記に延べるような移植者の特別な環境に合わせたコンスタントのBDS Cでコンパイルしなければならない．

#difin HC　　"S＊" /＊sはユーザのコンソールスクリーンにカーソルをもどす為に必要なストリング＊／

#difin CLEARS　"S" /＊sはユーザのコンソール上のスクリーンをクリアする為に必要なストリング＊／

BDSCIO. HとHARDWARE. HヘッダーファイルはTELEDがコンパイルされる前に，ターゲットコンピュータ機器構成に適切に編成されなければならない．

不必要なエディタコマンドは，ソースコードのエディタファンクションから一致したステートメントを取り去り，Teleditを再コンパイルすることにより可能である．

付録　K

CDB:BDS C　デバッガ

version 1.2
4 November 1982

David Kirkland
5915 Yale Station
New Haven, Connecticut 06520
(203) 787−9764

注記：　この付録は，BDS C または，BDS C ユーザーズ・グループから供給される CDB デバッキング・パッケージの解説書の一部分である．BDS C のディスケットの中に，これが含まれていないならば，本解説書は，CDB パッケージを購入するか否かを決定するのに十分な内容である．

K.1　はじめに —— 構成について

CDB は，BDS C コンパイラで書かれた，対話型のシンボリック・デバッガである．CDB は，プログラムに対して，ブレークポイントを設定する，トレー

スを行う，シンボリックな表示や，変数値の設定を可能にする．

これは，アプリケーション・プログラムの開発者が，プログラムの開発環境で使用するものとして，供給される．

デバッキング・パッケージは，3つの実行可能なファイルから成り立っている．最初のものは，L2.COMである．これは，オブジェクト・コードを生成するためのCRLファイルのリンカーであり，BDS C標準のCLINKリンカーの代替版である．本パッケージに含まれるL2リンカーは，Mark of the Unicon社のScott Layson氏によって書かれたL2リンカーを，多少修正したものである．この新しいL2リンカーは，新しいデバッキング機能とともに，Layson氏のL2のすべての機能と，多少のバグを取り除いたものである．L2は，デバッキング・パッケージの他の部分で使用するためのCOMファイルや，シンボル・テーブルを生成する．

2番目のものは，プログラム開発者（本解説書では，“ユーザー”と称している）が，デバッグを行うためのプログラム，CDB.COMである，CDBは，ユーザーが投入したコマンド行の引数を解析し，メモリー内のさまざまな，データ・テーブルを準備し，本パッケージの3番目のプログラムである．CDB2.OVLを起動する．CDB2は，CP/MのBDOSの直前のメモリーに常駐し，デバッグするプログラム（“ターゲット・プログラム”）を，TPA（“Transient Program Area”の略．通常，16進で0100のアドレスから始まる）からロードする．つまり，CDB2は，ターゲット・プログラムといっしょに，メモリー内に常駐する．CDB2がターゲット・プログラムをロードすると，実行を開始するために，ターゲットのメインルーチンへ処理を移す．ターゲット・プログラム内の(1)関数に入るとき，(2)関数から戻るとき，(3)コンパイルされたCの命令文の最初のコードを処理するときに，処理は，CDB2に移り，ターゲット・プログラムを続行したり，中断したり，中断してデバッガのコマンドに移ったりする（先に述べたように，列挙された出来事によって，いちいちCDB2が呼び出されるわけではない）．

本解説書では，角かっこ［　］で囲まれたものは，オプションであると解釈

される．したがって，その部分は省略可能である．

K.2　デバッガの作成

デバッガのいくつかのモジュールに対して，さまざまな変更が行なわれる可能性があるので，パッケージは，ソース・コードで供給する．このセクションでは，ソース・コードを，3つの実行可能なファイル，L2.COM，CDB.COM そしてCDB2.OVLに変換するための手順について解説する．

L2の作成

L2は，カスタマイズの必要性がないので，ユーザーが，自分自身のバージョンを作成する必要はない．供給されるL2は，現在ログされているドライブからC.CCCとDEFF *.CRLを取り込む．これを変更するには，L2.C内の説明に従って，# define DEF-DRIVEのマクロの宣言を変更する．もし，L2を変更(または，バグ取り)するならば，CDBパッケージに含まれているScott W. Layson氏による"The Mark of the Unicorn Linker for BDS C"の解説書に従って，L2.COMを作成する．しかし，これらの変更を行うときに，次のことに注意する必要がある．(1)使用する必要のあるファイル，SCOTT.Cがない．(2)他のバージョンのL2を用いて，L2のデバッガ・バージョンを作成するときは，CCに対して，"-e4800"を指定しなければならない．また，CLINKを用いて，L2をリンクするときは，"-e4c00"を指定しなければならない．その手順は，次のようになる．

```
CC l2.C -e4c00      [または，-e4800]
CC chario.c
clink l2 chario     [または，12 12 chario]
```

CDB2を記憶する場所

CDB.COMやCDB2.OVLを作成する前に，まず，CDB2.OVLを，メモリーのどこかに配置するかを決める必要がある．CDB2は，ターゲット・プログラ

ムと，それが使用するスタック・ポインタより上に，また，CP/M の BDOS と CDB2 自身のスタック・ポインタより下のメモリーに配置する．CDB2 は，0x4600 (18k) より少し少ない大きさで，外部領域は約 0x0980 バイトである．それから，CDB2 のスタック(BD08 の直前から始まる)の大きさを，約 ox0480 バイトに設定した．この値は，変数の値やシンボルの表示で（再帰的に）使用される複雑な式を，計算するのに十分な大きさであると思う．0x5400 になるので，CDB2 の開始アドレスは，BDOS の開始アドレスより，0x5400 だけ手前になる．私のシステムでは，BDOS の開始アドレスが 0xE406 なので，CDB2 の開始アドレスは，0x9000 になる(これは，これから述べる例の値として利用する)．（もし，あなたのシステムの BDOS の開始アドレスが，わからないなら，DDT を用いて，アドレス0005のジャンプ命令を調べることによって知ることができる．まず DDT を起動する．DDT は，“－”のプロンプトを表示するので，“L5”とタイプする．DDT が表示したリストの最初の行が，BDOS の開始アドレスであり，それは，最後が“06”で終っている．）

供給されるディスケットに含まれる CDB.COM と CDB2.OVL は，D300 以上の開始アドレスを持つ BDOS のシステムで動作する．60K バイト以上の RAM を持つほとんどのシステムで，このプログラムを使用することが可能であるが，ターゲット・プログラムとシンボル・テーブルの領域として 31K バイトしか割り当てられない．もし BDOS が D300 以上の開始アドレスを持ち，ターゲット・プログラムの領域をもっと必要とするならば，あるいは，BDOS が D300 以下のシステムならば，正しく動作するデバッガを作成しなければならない．

CDB2 を配置するアドレスが決定したならば，CDB.H ファイルの #define CDB2ADDR の宣言を，決定した値に変更する．ついでに，#define CDB2_DRIVE の宣言も変更するとよい．これは，CDB に対して－d オプションを指定しないときに，CDB2.OVL ファイルを，どのドライブからロードするかを指定するものである．ここには，“A”のようなドライブ名（コロンを付けずに)か，あるいは，“0/A”のように，ユーザー領域の指定子とドライブ名を含んだものを指定することができる．デフォルトでは，ドライブ名は指定されて

いないので，CDB2は，現在ログされているドライブのユーザー領域のディレクトリからロードされる．

CDBの作成

CDB.Hファイル内のCDB2ADDRの変更が済んだら，CDBの2つのモジュールを次のようにコンパイルする．

```
cc cdb.c   -e3200
cc build.c -e3200
l2 cdb build
```

CDB.SUBサブミット・ファイルは，上記の処理を一括して実行してくれる．

CDB2の作成

CDB2をコンパイルするには，CDB2の外部領域のアドレスを知る必要がある．外部領域は，CDB2のコードのすぐ後のアドレスから始まるので，CDB2ADDRの値に0x4600（K.2の"CDB2を記憶する場所"で述べられているコードの長さ）を加えた値を用いる．私のシステムでは，0xd600になるので，CDB2をコンパイルするときに，コンパイラに対して"-e d600"を指定する．

CDB2は，7つのソースファイルから成り立っている．それらをコンパイルするときには，次のように入力する．

```
cc cdb2.c    -exxxx
cc atbreak.c -exxxx
cc break.c   -exxxx
cc command.c -exxxx
cc print.c   -exxxx
cc parse.c   -exxxx
cc util.c    -exxxx
```

xxxx は，外部領域のアドレスである (D600 のように)．また，CDB2. SUB サブミット・ファイルを用いると，次のように入力するだけでよい．

```
submit cdb2 xxxx
```

xxxx は，外部領域のアドレスである．また，次の入力

```
submit cdb2 xxxx d:
```

では，デフォルトのドライブにソース・ファイルがないとき，ドライブ d からソース・ファイルを取り込むことを指示する．

すべてのCファイルをコンパイルしたら，次は，アセンブラのソース・ファイル，DASM. CSM をアセンブルする．これは，CASM フォーマットで記述されているので，CASM プリプロセッサを使用してからアセンブルする．次のように命令を入力する．

```
casm dasm
asm dasm
ddt dasm.hex
g0
save 3 dasm.crl
```

コンパイルされた CASM を持っていないユーザーのために，DASM. CRL ファイルを供給されるディスケットに記録してある．

最後に作成するファイルは，NULL. SYM と呼ばれる空のファイルである．CDB2. OVL は，オーバーレイ・セグメントとして作成されるので，L2 は，ルート・セグメントで定義された関数のシンボル・テーブルを読む必要がある．なぜならば，ルート・セグメントも，関数も存在しないのだが，L2 に対して，-ovl オプションを指定すると，ルートネームを必要とするので，次の命令

```
save 0 null. sym
```

を実行することによって，リンカーに対して，空の（つまり，ダミーの）ファイルを指定する．

さて，これで，すべてのCRLファイルが用意できたので，リンクを行なうために，次の命令を入力する．

```
12 cdb2 dasm atbreak command break print parse util
   -ovl null yyyy -wa
```

ここで，yyyy は，CDB2ADDR の値を16進で指定する．LCDB2.SUB サブミット・ファイルは，これらの処理を，一括して行ってくれる．

デバッガを使う準備は整った！

RST ベクタを変更する

供給されるデバッガは，ブレークポイントを生成するために，" RST6 " 8080 命令を使用する．RST6 命令に出会うと，処理(厳密にはプログラム・カウンタ)は，アドレス 0x0030 へと移る．あるシステムでは，このメモリーエリア(または RST6 命令自身）を他の処理に割り当てているものがある．もしそのような場合は，ブレークポイント機能で用いる RST ベクタを他の値に変更する必要がある．RST のベクタは，1から7までである．RST0 は使用できない．RST のベクタを変更するには，L2.C，CDB.H と，DASM.CSM を変更しなければならない．

L2.C と CDB.H は，#define RST-NUM の値を，適切な値に変更し，DASM.CSM は，RstNum EQU の値を，同じ値に変更する．値は，1から7までの範囲で指定すること．

最後に，ターゲット・プログラムを（−k オプションを用いて）コンパイルするときに，新しい RST ベクタの番号を指定する必要がある．−k オプションのかわりに-kn と指定する．ここで n は新しい RST ベクタの番号である．

K.3 デバッガの起動方法

デバッガを使用するには，まず，ターゲット・プログラムをコンパイルし，リンクしなければならない．それから，デバッガを起動する．このセクションでは，その方法について解説する．

コンパイル：　CCの－kオプション

BDS Cユーザーズ・ガイドに述べられていよにうに，－kオプションは，コンパイラに対して，(1) CDBという拡張名を持ったシンボル・テーブルを生成する．(2)コンパイルされたコードの中に，RSTの命令を挿入する，を指定する．ユーザーは，CCに対して，他のコンパイルと同様のコマンドと，－kオプションを付加したものを発行する．例えば，

```
cc target.c －k
```

リンク：L2の-D，-Sと-NSオプション

ターゲット・プログラムをリンクするには，CLINKのかわりに，L2リンカーを使用すること．L2の解説書で述べられているように，L2は，CLINKとは，異なるコマンド・シンタックスであり，デバッガ・バージョンのL2は，次のような，新しく追加されたオプションを含んでいる．

－D　CDBと互換性のあるモジュールを生成し，出力する．このオプションは，重要な関数の先頭に，RST命令を挿入する．－sや-nsオプションが，同時に指定されない限り，DEFF ＊.CRLの関数が，DEFF ＊.CRLの関数を参照するもの以外の関数の先頭に，RST命令を挿入する．

－S　－sの後に指定されるCRLファイルは，“システム”ライブラリとして，取り扱われる．システム・ライブラリ関数によって参照されるシステム・ライブラリ関数は，L2によってRST命令を挿入しないし，デバッガによってトレースされることもない．－sと-nsオプションを指定せず

に，－dオプションを指定すると，“－s diff diff2 diff3”と指定したのと同じものになる．

－NS システム・ライブラリ・ファイルを持たないことを指定する.このオプションは,DEFF ＊.CRL がシステム・ライブラリであるという，デフォルト解釈を無効にするために用いる．

CDB の起動

デバッガを起動するには，次の書式でコマンドを入力する．

cdb target－name [－l [local－cdbs] [－g [global－cdbs]]
[－d [user /] drive] [% [target operands]]

－l と－g オプションは，ターゲット・プログラムで使用される変数などの情報を含んだシンボルファイル .CDB を読み込む．－l －g を指定しないときは，デフォルトとして target-name. CDB が使われる．ターゲットのソース・コードが，2つ以上のファイルから成り立っているときに，これらのファイルのシンボルの定義を利用したければ，それぞれのソース・ファイルの CDB ファイルを CDB に介して与える必要がある．それぞれのソースファイルが，全てのグローバルを宣言するヘッダーファイル(H)を包含しているならば，－l オプションを使用するのみで，－g オプションを指定する必要はない．これらのオプションで指定するファイルネームのかわりに 0 を指定すると CDB は（ローカルか，グローバルの)，シンボル・ファイルを読み込まない．これらのオプションを，引数を付けないで指定すると，CDB はシンボル・ファイルの入力を要求するプロンプトを表示し，ユーザーがシンボル・ファイルのファイルネームを入力するのを待つ．空の行（単に CR）は，入力プロンプトを打ち切る．

“%”オペランドは，ターゲット・プログラムの引数を指定するものである．“%”以後に表われるオペランドは，ターゲット・プログラムの引数として扱われる．“%”の後に，オペランドが何もない場合，引数の入力を要求してく

る（CDBは，“％”の後に与えられた引数を，“argv”に渡さずに，例えば，0x0080からのメモリーに書き込む．そして，ターゲット・プログラムの中のC.CCCが，それを解析する）．

−dオプションは，CDB2.OVLをロードするドライブ(ユーザー領域も指定可)を指定する．供給されるパッケージのデフォルトは，CP/Mのデフォルト・ドライブからロードするが，これは，変更することができる．

標準的なCDBの起動は，次のようなものである．

```
cdb target
```

要約

標準的な，デバッガの手順は，例えば，target.cというプログラムを用いて，次のように行なう．

```
cc target.c −k
l2 target  −d
cdb target
```

さらに複雑な例をあげると，FOO.Cという“main”関数と，いくつかの関数が含まれるているソース・プログラムと，それが必要とする関数が含まれるているBAR.CとLIB.Cがある．FOO.Cは，同じグローバル変数をもっている（GLOBAL.Hというヘッダーファイルが#includeによって包含される）．LIB.Cは，グローバル変数を使用しないライブラリ関数の集まりである．最後に，STBLIB.CRLファイルという(デバッグされた)ファイルを使用すると想定する．これをコンパイルするには，次のコマンドを入力する．

```
cc foo.c −k
cc bar.c −k
```

cc lib.c −k

次に，リンカーを用いて，これらのプログラムをリンクし，FOO.COM を作成する.

l2 foo bar −l lib −s stdlib

−s オペランドは，L2 に対して，STDLIB.CRL ファイルに含まれる関数をトレースしないように，FOO.COM を生成することを指示する．デバッガを起動するには，次のように入力する.

cdb foo −l bar lib

−l オペランドは，CDB に対して，CC によって作成された BAR.CDB と LIB.CDB のシンボル・ファイルに含まれるすべてのローカルなシンボルをロードするように指示する．FOO.CDB に含まれるローカルおよびグローバルのシンボルは，すべてロードされる.

K.4 デバッキング・コマンド：デバッガの使いかた

このセクションでは，機能別にまとめた CDB のコマンドについて解説する.

デバッガが起動すると，CDB2 の配置アドレス(CDB2ADDR と同じ)，ローカルとグローバルのシンボル・テーブルの大きさ，そして，ターゲット・プログラムのスタック(CDB2 のプログラムのすぐ下)の値を表示する．そして，ターゲット・プログラムに処理を移す．ターゲット・プログラムは，C.CCC 内の初期化ルーチンから，“main”関数までのコードを実行する．“main”関数のエントリには，ブレーク・ポイントがセットされているので，処理は，デバッガのコマンド・レベルへと戻る.

ブレーク・ポイント

CDB は，ターゲット・プログラムに対して，文単位で実行させることができる．ターゲット・プログラムの実行を中断するには，ブレーク・ポイントとキ

ーボードからの割り込みの2つの方法がある．ブレーク・ポイントを設定すると，ターゲット・プログラムのCの命令文を実行する直前に，実行が中断される．キーボードからの割り込みが発生すると，ターゲット・プログラムの，次のCの命令文を実行する前に，実行が中断される．キーボードからの割り込みは，単に，キーをタイプするだけで，CDB がこの文字を取り込む時に，実行が中断される（しかし，ターゲット・プログラムが，キーの入力を待っているようなときは，割り込みは，発生されない)．

ブレーク・ポイントを設定するには，“break”コマンドを投入する．

```
b [reak] [function-name] [statment-number [count]]
```

角かっこで囲まれた部分は，省略可能である．“break”コマンドは，単に“b”と入力してもよいし，“br”，“bre”なども同様に受け付ける．また，function-name と statement-number も省略可能である．) function-name を省略すると，ブレーク・ポイントは，現在の関数の statement-number で指定される命令の文番号に設定される（現在デバッグしている関数の名前は，ターゲット・プログラムを中断した後に表示される．また，それは，“list”コマンドを利用することによって見ることができる)．statement-number は，指定した関数内の正確なブレーク・ポイントの場所を指定する．命令文は，関数の先頭（関数宣言の左かっこが含まれる行）を1とする行番号によって，行単位で番号付けされる．次の例ように，

```
a=5;putchar ('x'); while (* s) s++;
```

1つの行に，複数の命令文が含まれているもののどれかを指定するには，その行番号nの最初の命令文は，n.0，次は，n.1 というようにする．(上の例で，行番号が7だとすれば，“a=5;”は，7.0“putchar (x);”は7.1“whil (* s)”は7.2，そして，“s++;”は，7.3になる．) 10進数が与えられなければ，“0”と解釈する．文番号は，次のように定義する．

```
sn:=line-number [.statment-number-within-line]
```

（ここで，line-number は，文番号を，statment-number-within-line は，行の中の文番号を表わす.）

CC は，時として，ソース・コード，またはそこから生成されるコードを，再アレンジすることがある．このような場合，ユーザーが指定する命令文に，ブレーク・ポイントを設定するのは，困難になる．CC がこのような，"特別な命令"を生成するのは，次のような場合である．

(1) コンパイラが，ループ命令（"while"，"for"や"do"）を処理するときは，ループの最後の部分に，条件分岐命令を生成する．

(2) "for"命令文の"再初期化式"（"for"命令文の 3 番目の式）はループの最後の部分におかれる．このように，"for"文の後の式は，ループの最後の文の文番号の次の値になる．

話はそれるが，特別な文番号として，0 と－1 がある．文番号 0 は，関数のエントリで，関数のコードが実行される前に位置する．文番号－1 は，関数から戻る場所で，関数から戻った直後に位置する（そして関数からのリターン値を表示する）．

これまで，COUNT オペランドについては，何も触れなかった．"break"コマンドで指定されるブレーク・ポイントは，COUNT で指定された回数だけ，ブレーク・ポイントを通らない限り，ターゲット・プログラムは中断されないのである．COUNT のデフォルト値は，1 であり，したがって，設定されたブレーク・ポイントに 1 度でも達すると，ターゲット・プログラムは中断される．COUNT を指定するには，文番号（statment-number）も必ず指定しなければならない．

ブレーク・ポイントは，1 度に40か所まで定義することができる．

ブレーク・ポイントを，取り除くには，"reset"コマンドを使用する．文法を次に示す．

r [eset] [function-name] [statement-number]

デフォルトは，"break"コマンドと同じである．ブレーク・ポイントでない

地点を指定すると，エラーになる．“clear”コマンドは，全てのブレーク・ポイントをリセットするために使用する．文法は，次のとおりである．

clear

（“clear”コマンドは，フルスペルで入力しなければならない．）

“list breakpoints”コマンドは，設定されているすべてのブレーク・ポイントを表示する．

コードの実行

コンパイルされたCのコードを実行するためのいくつかのコマンドについて解説する．“go”コマンドは，単に（前に中断された地点から）実行を開始し，ユーザーが，キーボードから割り込みを行うか，ブレーク・ポイントに達するまで続けらる．このコマンドのオペランドは，ない．

ターゲット・プログラムの実行を，モニタしながら行うには，“trace”コマンドを用いる．文法は，次のようになる．

t [race] [number-of-statements]

デバッガは，number-of-statementsで指定された数の命令文をトレースし，命令文を実行する前に，それが含まれる関数名と，文番号を表示する．実行される命令文が，number-of-statementsに達する前に，ブレーク・ポイントや，キーボードからの割り込みがあると，実行は，その場所で中断される．number-of-statementのデフォルトは，1である．

“untrace”（“walk”としても知られる）コマンドは，“trace”コマンドに類似しているが，命令文を実行する前に，関数名と文番号を表示しない．文法は，次のとおりである．

u [ntrace] [number-of-statements]

“trace”と同じように，number-of-statementsで指定される数の命令文を

トレースし，ブレーク・ポイントや，キーボードの割り込みがあると，実行が中断される．number-of-statementsのデフォルトは，1である．

"run"コマンドは，CDBからターゲット・プログラムへ処理を渡し，デバッガを離れる．したがって，"run"コマンドを実行した後で，デバッガに処理を戻すことはできない．また，"run"コマンドには，省略名はない．

変数の表示

"dump"コマンドは，メモリーの内容を表示する．コマンドの文法は，次のとおりである．

d [ump] expression [multiple] [format]

"dump"コマンドは，"p [rint] "や"，"（カンマ）としても用いることができる（同義語）．

"dump"コマンドは，expressionによって指定されたメモリーの内容を表示する．expressionの全ての定義は，後述するが，最も一般的な使い方として，変数名（" i "や"foo"など）や，数値（0x0100や43000など）をexpressionに指定するであろう．変数名や，数値がexpressionに指定されると，CDBは，その変数の宣言や数値に適したフォーマットで表示する．もし，変数が構造体ならば，CDBは，その構造体に含まれるエレメントも含めて表示する．ユーザーがformatを使用するのは，expressionに指定する式が，変数や数値ではなく，整数のアドレスであるようなときであり，formatに指定できるのは，次のものである．

c	文字
p	ポインタ
iまたはw	整数／ワード
s	文字列（NULLで終結される文字配列）

formatのデフォルトは" w "である．

multipleオプションは，表示するメモリーの大きさを指定する．"dump"コマンドは，指定されたformatをmultipleで指定した数だけ表示する．例えば，

```
dump 0x0100 10 c
```

は，0x0100から10文字，つまり0x010Aまでを表示する．また，

```
dump 0x0100 10
```

は，formatが指定されないので，デフォルトの"w"と解釈して，0x0100から0x0114（20バイト）までの内容を，ワードで表示する．

expressionの文法は，次のようになる．

```
expression :=   *式
                                        一次式
    一次式      :＝整数
                  識別子
                  （式）
                  一次式［式］
                  一次式，識別子
                  一次式－＞識別子
```

CDBの式は，基本的に，Cの式のような論理演算子や算術演算子を，含んでいない．式は，次に示すように，かなり複雑になりうる．

```
table［table［1, i］, j］. name［10］
```

"dump"コマンドの表示を止めたいときは，任意の文字をタイプする．

Cの有効範囲の規則は，シンボルの参照のために用いられる．これは，例えば，デバッガが，fooという関数のブレーク・ポイントに達し，中断したときに，もしbarという変数が，その関数内で定義されているなら，変数barは，その関数内のものを表わすことになるが，その関数内で定義されていないときは，

グローバル変数"bar"を参照しようとする．これは，有効範囲の規則によって，Cの関数内で定義される変数は，グローバル変数の名前と同じであってはいけない．CDBは，この有効範囲を無視することができる．グローバル変数を指定するときは，その変数名の先頭にバック・スラッシュ（"＼"，ただし，日本の多くのパーソナルコンピュータは，"¥"コードに割り当てられている）を付加すればよい．例として，"foo"関数から，グローバル変数"foo"を指定するには，次のように入力する．

```
dump \foo
```

"dump"コマンドは，シンボルの値ではなく，アドレスを指定することもできる．この場合は，expressionに指定する式の前にCの"アドレスを求める"算術子"&"を付加する．例えば，"table"という変数のアドレスを求めるには，次のように入力する．

```
dump &table
```

また，次のような，複雑な式を使用することもできる．

```
dump &table [i, j]
```

変数の値の設定

"set"コマンドは，メモリーに，データをストアする．コマンドの文法は，次のようになる．

```
s [et] expression value [c]
```

expressionによって指定されるアドレスに，値valueを書き込む．値は，通常16ビットであるが，次のようなとき，8ビットとして書き込まれる

(1) expressionが，char型の変数のとき
(2) expressionが，'#'のように，引用符で囲まれているとき
(3) "c"オプションを指定したとき

リスト・コマンド：　さまざまなインフォメーション

“list”コマンドは，次のようにさまざまなインフォメーションを，表示する．

l [ist]	現在の関数名と文番号を表示する．
l [ist] a [rguments]	現在の関数の引数を表示する．
l [ist] b [reakpoints]	すべてのブレーク・ポイントを表示する．
l [ist] g [lobals]	すべてのグローバル変数を表示する．
l [ist] l [ocals]	現在の関数内で宣言されている変数を表示する．
l [ist] m [ap]	ターゲット・プログラムのリンク・マップを表示する．
l [ist] t [raceback]	main から現在の関数までのトレースされた関数を表示する．

“list globals”や，“list locals”による変数の表示を止めるには（キャリッジ・リターン以外の），任意のキーをタイプする．大きな配列の表示をスキップしたいときは，キャリッジ・リターンをタイプする．

終了コマンド

デバッガ作業を終了し，CP/M に戻るためには，“quit”コマンドを使用する．

このコマンドは，フルスペルで入力しなければ，受け付けられない．

K.5　デバッガのコマンドのリスト

文番号の定義

文番号：＝［行番号］［．行の中の命令文の番号］

式の定義

式　　　：＝　＊式
　　　　　　　一次式

一次式　：＝　整数
　　　　　　　識別子
　　　　　　　（式）
　　　　　　　一次式［式］
　　　　　　　一次式．識別子
　　　　　　　一次式－＞ 識別子

b［reak］［関数名］［文番号［回数］］

ブレーク・ポイントを設定する．デフォルトは：

関数名	現在の関数
文番号	0
回数	1

clear

全のブレーク・ポイントを取り除く

d［ump］　式　［multiple］［format］

式の結果を“format”の書式で，“multiple”回くり返す．デフォルトは：

multiple 1
format “i”または,シンボルの宣言による.
同義語　p［rint］，（カンマ）

“format”の指定は，

c	文字

p	ポインタ
iまたはw	整数／ワード
s	文字列

g [o]

　実行を開始する.

l [ist]

　現在の関数と文番号を表示

l [ist] a [rguments]

　現在の関数の引数を表示

l [ist] b [reakpoints]

　全てのブレーク・ポイントの表示

l [ist] g [lobals]

　全のグローバル変数の表示

l [ist] l [ocals]

　現在の関数内で宣言される変数の表示

l [ist] m [ap]

　ターゲット・プログラムのリンク・マップを表示

l [ist] t [raceback]

　トレースされた関数を表示

quit

CP/Mに戻る

r [eset] [関数名] [文番号]

ブレーク・ポイントを取り除く．デフォルトは，

関数名　　現在の関数

文番号　　0

run

実行を開始する．デバッガへは戻らない．

s [et] 式　値 [c]

値をメモリーに設定する．値は16ビットとして扱われるが，次のようなとき，値は8ビットとして扱われる．

(1) 式がchar変数のとき．

(2) 式が引用符で囲まれているとき．("#"のように)

(3) "c"オプションを指定したとき．

t [race] [回数]

トレース．実行された命令文を表示する．回数のデフォルトは1．

u [ntrace] [回数]

トレース．ただし，実行された命令文を表示しない．回数のデフォルトは1．同義語としてw [ork] がある．

BDソフトウェアC　コンパイラ V1.50aのための追加インフォメーション

Leor Zolman
BD Software
P.O.Box 9
Brighton, Ma.02135

この文書は，BDS C V1.50aに関する追補の文書である．初めのセクションで，ソフトウェア（供給されたもの）と，コンピュータ・システム間の，非適合性について述べる．次のセクションで，コンパイラとライブラリの新しい機能について述べ，最後のセクションでは，V1.50で発見されたバグについて述べる（これらのバグは，V1.50aでは取り除かれている）．

システムとの非適合性

次に述べることは，V1.5のユーザーズ・ガイドで述べられていない，コンパイラのコンフィグレーションや操作に関する事である．

- V1.50のCDBデバッガと，C.CCCランタイムパッケージには，危険な矛盾が存在する．BDS Cのプログラムを，デバッガの下で走らせる場合，CCコンパイラに対して，オプション－kを指定する必要がある．これは，プ

ログラムをデバッグするために，コンパイラによって生成されたコードの中のさまざまな場所に"RST6"（デフォルト）の命令を挿入するためである．CDBデバッガが起動すると，RST6のベクタ・エリア（CP/Mのベースページ内のロケーション30h）からの数バイトに，ある命令を書き込む．もし，RST6のベクタを，なにかの割り込みハンドラーで使用しているなら，CDBは，このベクタを書き換えてしまうので，システムが正常に動作しなくなってしまう．また，V1.50のBDS Cランタイム・パッケージ（C.CCC，ソースはCCC.ASM)は，**－k**オプションを指定してコンパイルされたプログラムを，単体（CDBが，メモリーに存在しない）で走らせることを可能にするために，RST6のベクタを書き換えてしまう．したがって，RST6のベクタを使用しているシステムでは，Cでコンパイルされた**どんなプログラムも**実行することができない．

V1.50aのランタイム・パッケージは，新しいオプションとして，RSTのロケーションの書き換えを行うか否かを指定することができる．したがって，**－k**オプションを用いてコンパイルされたプログラムを，単体で実行するときに，RSTのベクタを書き換えればよい．また，どのRSTベクタを使用するかを**任意**に指定することができる．ランタイム・パッケージのデフォルトの設定は，システムの異常動作を招かないように，**すべての**RSTロケーションを**書き換えない**．

もし，あなたのシステムが"RST6"のベクタを使用していなければ，CCC.ASM内の"USERST"のシンボルを，"TRUE"に定義し，CCC.ASMを再びアセンブルし，**－k**オプションを用いてコンパイルされたプログラムを単体で走らせることを可能にするため，新しいC.CCCをつくればよい．

もし，あなたのシステムが"RST6"のベクタを使用しているならば，CCC.ASM内の"RSTNUM"のシンボルを，空いているベクタ番号に変更し，"USERST"のシンボルも，先に述べたように変更し，再びアセンブルして，新しいC.CCCを生成する．そして，新しく設定されたRSTのロケーションを使用するCDBを作成する（CDBを作成する方法は，CDB

解説書を見よ）.

あなたのシステムで，空いているRSTベクタがない場合，CDBを走らせることはできない.

- biosとbdosのライブラリ関数は，CP/MのBDOSが用いるファンクション・コールからのリターン値を用いることを前提としている．これらは，すべてのCP/Mシステムで動作するが，CP/Mに準拠したシステム（SDOS，CDOSなど）では，この限りではない．もし，このようなシステムを使用していて，正しく動作しないファンクション・コールがある場合は，あなたのオペレーティング・システムのBDOSが，HLに値を戻しているか，BIOSがAに値を戻しているか…などを調べる必要がある．もし，それが，標準CP/Mの仕様と異なるものであるなら，bdosあるいは，bios関数を正しい動作をさせるために書き直さなければならない.

- BDS Cのbiosライブラリ関数は，システムの0000h番地（CP/Mのベースページの最初の命令）に書かれているBIOSジャンプテーブルの2番目のエントリ（wboot）にジャンプする命令を基にしている．このようなシステム以外（たとえば，Xerox820）では，bios関数は，正しく動作しないので，BIOSジャンプテーブルを正しく参照できるように，プログラムを書き直さなければならない.

新しい関数と特徴

このセクションでは，ユーザーズ・ガイドで述べられていない新しい関数と，特徴について解説する.

1. 次のライブラリ関数が，標準ライブラリに加えられた.

```
int fappend (name, iobuf)
char * name;
FILE  * iobuf;
```

この関数は，指定されたファイルが存在した場合，そのファイルの EOF からアペンド（追加）するために，オープンする．この関数は，テキストファイルにのみ使用される．

```
lprintf (format, arg1, arg2, …)
char * format;
```

この関数は，printf 関数と同じ機能を有するが，その出力は，CP/M の " LIST " 装置へ送られる．

```
int index (str, substr)
char * str, * substr;
```

str の文字列の中に substr の文字列が含まれていたら，その位置を戻す．見つからないときは，－１を戻す．

```
int memcmp (source, dest, length)
char * source, * dest;
unsigned length;
```

source と dest から長さ length バイトのメモリーを比較し，完全に同じならば真(１)を，同じでなければ偽(０)を戻す．この関数は，アセンブリ言語で記述されているので，高速である．

2. パッケージに含まれる．CP.C ファイルコピーユーティリティは，新た

に“ベリファイ”オプションを含んでいる．詳しい使い方は，CP.Cのソースコードを参照せよ．

3. 新しいサンプル・プログラムDI.Cが含まれている．これは，簡単な，ファイル比較ユーティリティで，2つのファイルを高速に比較し，差異を表示する．

4. TELEDIT.Cテレコミュニケーション・プログラムが改良された．ユーザー間のコミュニケーションを簡単にし，ファイル転送の技法を，簡潔にした．

修正されたバグ

次に述べるのは，V1.50に存在したバグのリストである．**これらは，V1.50aでは，取り除かれている．**

1. CCとCLINKのデフォルト・ドライブに関する問題で，あるコンパイル，リンク条件のもとで，悪い論理ドライブをアクセスしてしまう．例えば，CC.COMで用いるデフォルト・ドライブを，ドライブCに設定したとする（セクション1.9.2.1で述べられている）．
現在ログされているドライブがAだとして，次のコマンドを用いて，ドライブBのプログラムをコンパイルしたとする．

```
A> CC b:test.c
```

処理が始まり，test.cに含まれているデフォルトのライブラリ領域を参照する# include 命令（ファイルネームが＜　＞で囲まれている．）に出会うと，デフォルト・ドライブであるAを検索せずに，ドライブBを検索してしまう．この種の問題は他にもあるが，これらは，V1.50aでは，発生しな

いようである．

2. V1.50 の最も初期のもので，構造体の宣言を，2 度使えないものがあった．例えば，次の文，

```
struct foo {
    int a;
    int b;
} ;
struct foo foostruct;
```

では，foostruct を宣言する部分で，" illegel identifire " のエラーメッセージが出力される．

3. CLIB プログラムで，ディスク名を付けたファイルネームは，6 文字までしか，認識されない．

4. 定義のない定義文があると，コンパイラは，動作しなくなる．例えば，次の命令など．

```
# define BLANK      /*定義がない*/
```

5. 関数の宣言の間に，ホワイトスペースを入れると，宣言の処理ルーチンを混乱させる．次の例では，関数は，正しく宣言されない．

```
char *        /* 通常，この 2 つの行は，1 つにされる./*
func (n)      /* この方法は，正しいのだが正しく動作しない．*/
int n;
```

6. alloc と free 関数は，不意に，メモリー容量オーバー (Out of memory) を起こす．特に，小さな大きさの割り当て領域の間の大きな割り当て領域を解除するときに発生する．このバグは，解除された領域を再び割り当てるときに，その領域の最後からではなく，最初から割り当てることによって取り除かれた（この修正をしてくれたウィリアムC．コーレーIIIに感謝する）．

7. fprintf と printf 関数のあるバージョンで，ニューラインを正しく操作しないものがある．これは，低レベルのライブラリ関数である“ _fputc ”が正しくないからである．このバグの内容は，ニューライン記号を，ラインフィードを付加しない，単一の‘＼r’(ASCII コードで 0Dh) に変換してしまうというものである．

8. printf 関数のような，書式付き文字列を扱う関数で，書式付き文字列の最後の文字が%だと，正しく動作しなくなる．

9. #include 文を操作しているときに発生するエラーメッセージの行番号は正しくない．

10. CLINK で，（－v オプションを用いて）オーバーレイセグメントを作成しようとすると，次のエラーメッセージが出力される．

```
Warning! Externals extend into BDOS!
Warning! Exterals Overlap code!
```

このように表示されても，それが正しくない（つまり，エラーではない）ことがある．

あとがき

CP/M-80用に早い時期から安く販売されていたBDS Cコンパイラーは，米国内のユーザーズ・グループの協力により，より使いやすく強力になり，現在（1984年5月）バージョン1.50aとなっています．

私がライフボート社にマニュアルの和訳のことで話をした頃は，まだバージョンが1.46のときでした．

『近くバージョンが大きく変わるので，新しいマニュアルを訳してほしい』とライフボート社の社長・田先氏より連絡があり，数週間後まだ製本されていないサンプル用にタイプした版のコピーが送られてきました．

また，ユーザーズ・グループ発行のニュースレター，ディスケット数10枚も，後に私のところに届きました．田先氏が自分の足で集めてこられた国内には始めての資料でした．

マニュアルの直訳は，約3か月かかり，完成の頃には，日本国内でもバージョン1.50版が発表されその後，数か月に100本も売れるベスト・セラー商品となりました．

マニュアルは，直訳後，実際の使用の手引きとなるように，わかりやすい和文にする必要がありましたが，この時点から進まず約1年が過ぎてしまったわけです．今回，工学図書㈱より出版できることを喜ばしく思っています．タイプ版の誤字の多い英文，150ページ以上にわたり翻訳に協力していただいた久保明子嬢には心から感謝しています．

本来なら，ライフボート社から使用マニュアルとして発売される性格のものを，出版物として発行するように勧めた私が，最後まで自分の手で行なえなかった事を心苦しく思っています．

この和文マニュアルの出版により，多くの人が実際にC言語を使用され，多くのソフトウェアが発表されることを願っています．

1984年5月

渡　辺　　修

索　　引

《欧　文》

A

B

C

D

E

F

G

H

I

《和　文》

お

か

き

こ

さ

し

せ

て

な

ふ

ゆ

ら

ろ

わ

BDS C の使い方

昭和59年7月1日　初　版

検印省略

訳　者　稲川幸則
　　　　渡辺　修
発行者　笠原洪平

発行所　工学図書株式会社

（営業所）　東京都千代田区三番町5
郵便番号 102
電話　東京（262）3772番
振替　東京13465番
印刷所　㈱サンヨー—03(294)4951

定価　2,300円

ISBN 4-7692-0129-X-C3058